The Design of a High-Integrity Disk Management Subsystem

Oey, M.A.

2005

document version
Publisher's PDF, also known as Version of record

The Design of a High-Integrity Disk Management Subsystem

Typeset with LaTeX2e
Cover design by Tsjong-Won Bruinsma and Michel Oey
Printed by PrintPartners Ipskamp, Enschede, the Netherlands

Author's address:
Department of Computer Science
Vrije Universiteit Amsterdam
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
michel@cs.vu.nl
http://www.cs.vu.nl/~michel/

VRIJE UNIVERSITEIT

The Design of a High-Integrity Disk Management Subsystem

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 17 mei 2005 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Michel André Oey

geboren te Amsterdam

promotor:	prof.dr. A.S. Tanenbaum
copromotor:	dr. W. de Jonge

to my parents

Contents

Preface

The dissertation that lies in front of you is the result of many years of work. In this dissertation, I have tried to explain the conceptual ideas behind the Logical Disk as clearly and comprehensibly as possible. As is so often the case in research, simple ideas whose main concepts can be expressed in just a few sentences, turn out to be more complex when worked out in more detail. Especially, explaining a complex structure clearly requires much effort, hence the volume of the dissertation. Here, I would like to thank all who have helped and supported me over the years.

First and foremost, I would like to thank my supervisor Wiebren de Jonge and my promotor Andy Tanenbaum. Both have spent many hours reading and commenting on every page I wrote. I am indebted to Wiebren for suggesting that the Logical Disk project would make a good research subject for me. My relationship with Wiebren has been a pleasant one. I am very glad that I have come to know Wiebren both on a professional and on a personal level. I have never met a man who could so easily master any subject, and unravel any complex problem into clear concepts. I hope I have mastered those abilities to some extent as well. I am indebted to Andy Tanenbaum for giving me the opportunity to do research, and thank him for the patience he had with me. Thank you both for your guidance, support, and your confidence in me.

I would also like to thank Philip Homburg for all the effort he has put into the LD-project. Philip has contributed many ideas, most notably on the subject of cleaners and reorganizers, and has implemented the prototype of LD, and ran most of the experiments. Furthermore, I am thankful that he was willing to share his office with me for a few months. I have enjoyed our conversations on many topics, ranging from research to baroque music.

In the early stages of the LD-project, Reind van de Riet, Hans van Staveren, and Gregory Sharp also worked on the project. I enjoyed working with them, even though it was only for a short time. I want to thank all three of them for their contributions.

I need to say a special thank you to Gerco Ballintijn. Not only did he help and motivate me during the difficult stage of writing the dissertation, he also read every chapter and provided me with many helpful comments. I also appreciate his efforts to introduce me to the field of human psychology, although at times, it felt like I was the subject in one of his psycho-analyses.

Furthermore, I want to thank the members of my reading committee and the promotion committee, including Frans Kaashoek, Mary Baker, Herbert Bos, Thilo Kielmann, John Romein, Tim Rühl, and Reind van de Riet. They provided me with useful comments on

my dissertation.

Arno Bakker, Klaas Pereboom, and Barbara Söhngen have helped me with my Dutch summary. Especially Barbara's thorough comments improved it a lot. Thank you for putting in all the effort.

Credits for the design of my front cover go to Tsjong-Won Bruinsma. Even though he is a very busy man and I asked his help when the deadline was very near, he still put in a lot of time and has done a remarkable job. I am very grateful for all your help! The hard disk on the cover was graciously donated by Jan van der Weide.

Thanks also go to Stefan Blom who wrote the scripts that simplified processing the LaTeX 2ε source, and to Clemens Grabmayer and Michel Klein who gave me last-minute advice in the final stages of sending the dissertation to the printing-office.

At this point, I would like to switch to my friends who have played an imported role in another part of my life: music. During difficult times of research, I could always find strength and relaxation in making and listening to music. I was lucky to have been a member of several orchestra's, such as the VU symphony orchestra, the VU chamber orchestra, and the Naardens chamber orchestra. During those times I made many friends, many of whom I still see on a regular basis. There are far too many friends to mention them all here, but I want to thank all who have supported me and have shown interest in my work.

Thanks are also due to my fellow string quartet members including their spouses: Tsjong-Won & Michelle Bruinsma, Martine & Arnaud Jansen, and Marit van Vonderen. Since the foundation of our group in 1995, we have become very close friends. They have always supported me, and I am very happy that I can finally show them the end result. It is now my turn to support both Martine and Marit on the paths to their promotions. Good luck!

At the university, I enjoyed spending the lunch and coffee breaks with my fellow colleagues. I particularly enjoyed the entertaining discussions I had with them. On many occasions, a healthy dose of (absurd) humor was required to follow our discussions. Unfortunately, the composition of our lunch group changed so much over the years, that there are too many names to list them all, but I thank you all! However, I do want to mention a few friends with whom I have spent the most time: Chris & Maartje Niekel, Jaap Kreijkamp, Gerard Kok, Mirna Bognar, Jerry den Hartog, Paulien de Wind, Jeroen Ketema, Gerco Ballintijn, Arno Bakker, David Bouman, Sylvia van Borkulo, Ernst Verhoeven, and Niels Veerman. They have waited long enough for this moment to arrive. Well, my friends, now is the time to celebrate!

Over the years, I had the pleasure of sharing an office with a number of people. I want to thank Frank Dehne explicitly, since we have shared a room the longest. In Frank I had a quiet roommate, but not so quiet that the silence felt awkward. It was a pleasant environment to work in. Moreover, I could always ask Frank for help concerning many topics, including translating emails from Spanish.

Two great friends I have are Gerard Kok and Mirna Bognar. We have spent many afternoons and evenings together talking about all possible aspects of life, usually while enjoying good food and drinks. My friendship with Gerard goes a long way back. I can always rely on Gerard to advice me on any issues I have with computer hardware or software. Gerard, we should schedule another '*avondje bieren*' soon! Since I met Mirna,

I have come to know her as a person with a cheerful demeanor and the ability to make and especially maintain good friendships. Thank you both for your friendship and support, and I would like to include Stan van Gisbergen into my thanks as well. I truly hope we will be able to keep our friendship as close as it is now.

Now the time of the PhD-defense draws near, I am glad I will have the support of both Gerard Kok and Gerco Ballintijn. They have both graciously agreed to act as '*paranimf*.

I also want to thank Barbara Söhngen, who has supported me throughout the stressful last few months. I do not know many people that are so kind and loving as she. Dear Barbara, I am so thankful that you have become part of my life.

At the end of this preface, I want to thank my mother and brother for all the love and support they have given me over the years. It has been a long and arduous road, and without them it would have been even more difficult. Thank you for being there when I needed it.

Michel Oey
March 2005, Amsterdam

Chapter 1

Introduction

In this day and age, computers are everywhere. The advent of computers has made it possible to store and manipulate large amounts of information efficiently. In the beginning, computers were used only by large companies and institutions to store their data, such as information on their inventories, orders, customers, etc. In the last decade or so, however, the computer has spread enormously, and currently, computers can be found in nearly every household in the industrialized world. Advances in technology, which have increased the computer's processing power and its storage capacity, have made it possible to store ever increasing amounts of information. For example, video and audio (multimedia) applications, which often manipulate large data objects, are common nowadays.

Since the storage of data is a main task of computers, it is important that it is done correctly and efficiently. The correctness of data storage concerns the integrity of the data that are stored. In other words, the data stored by the computer should be stored persistently and safely. For example, a system failure should not cause part or all of the stored data to become inconsistent or even lost. The efficiency of data storage refers to performance; accessing the stored data should be fast. As computers become more powerful, they are used to process more and more data, and consequently, the performance of data storage is important for the overall performance of a computer system. Examples of techniques that are used to increase performance are caching, clustering of data on disk, avoiding or removing fragmentation on disk, etc.

However, though we agree that performance is important, we believe that correctness of data storage is even more important. Users entrust a computer with their precious data, and therefore, the computer is made responsible for protecting the integrity of these data. We, therefore, advocate that the quality aspects of storage, such as data integrity, should get more emphasis and should be improved. Unfortunately, this view is not generally shared by those who build computers and write software for them. Often the balance between performance and data integrity is tipped in favor of performance. For example, the popular file system of Linux, the Ext2 file system [Ext2FS; Card et al., 1994; Beck et al., 1998], performs well, but it does not guarantee any data consistency after a crash.

Another example that shows that performance is often considered more important than data integrity is that operating systems such as Linux and FreeBSD, by default, enable

'write caching' on the hard disk. Indeed, write caching can improve a disk's performance considerably. With write caching enabled, the disk acknowledges a write command as soon as the data have been stored in the disk's internal cache. The disk will write the data from the cache to the actual platters of the disk at a later point in time. Since writing data into the cache is much faster than actually writing data on the physical platters, the acknowledgment can be sent sooner with write caching enabled.

On the downside, with write caching enabled a disk's acknowledgment for a write command does not mean that the data are persistently (i.e., safely) stored on disk. Consequently, if a crash occurs, even data that the disk has reported as stored, may still get lost because during a crash all data in the disk's internal cache are lost. Furthermore, with write caching the order in which the data are written to the actual platters of the disk can be different from the order in which the data were acknowledged by the disk. This reordering may also lead to unexpected results and inconsistencies after a crash.

The main medium for on-line storage of large amounts of data in computers has long been the hard disk. (A detailed discussion of hard disks is given in Section 2.1.1.) This dissertation focuses on improving the usage of the hard disk as storage medium within the context of a single-disk storage system. In particular, we present ideas on how to improve the modularity of software that uses disks, and on how to improve the way data are stored on disk in order to improve data integrity without having to pay much in terms of performance. Thus, our main focus is on improving the quality of the use of disk storage.

In this dissertation, we focus on the small-scale use of hard disks within storage systems, such as the personal computer at home. Furthermore, we focus on maintaining data integrity in the face of system failures, including power failures. Already within the context of the personal computer, loss of data stored on disk can be disastrous to the owner of these data. Unfortunately, the average home user does not take extensive precautions, such as making regular backups, to be able to recover from the loss of data due to system failures. Therefore, in our research we try to improve the quality of disk storage in the context of a single-disk system to prevent losing data due to system failures.

This chapter provides an overview of the dissertation. It starts by presenting a brief introduction to hard disks as the main storage medium in Section 1.1. In Section 1.2, we identify three types of problems with current disk usage. Section 1.3 presents some of the existing techniques that are used to solve one or more of these problems. Section 1.4 then introduces the Logical Disk, a new storage system, that tries to overcome the problems mentioned in Section 1.2. Section 1.5 lists the contributions of this dissertation. Section 1.6 closes this chapter by providing an outline of the remainder of this dissertation.

1.1 Disk Storage

In computers, the **hard disk**, or simply **disk**, has been the dominant device for storing large amounts of data on-line. The term 'on-line' in this context means that the data are readily accessible to the computer. The hard disk is very suitable for on-line storage because it has the following characteristics:

(1) Large capacity — currently, single off-the-shelf hard disks can already store over

200 GB[†], and this number is rising quickly.

(2) Persistent storage — in contrast to RAM (main memory) and similar to NVRAM (nonvolatile RAM), disks can store data persistently, which means that the contents of the disk are maintained if power is cut off from the disk (for example, when the computer is turned off).

(3) Efficient random access — although not nearly as fast as RAM or NVRAM, disks do provide fairly fast random read and write access to their contents. In contrast, tapes only allow slow sequential access. CDs and DVDs can also provide random access, but their speed is low compared to the speed of hard disks.

(4) Cost — the price per MB is low and is still dramatically decreasing. Furthermore, the price per MB for disks has long been and still is two orders of magnitude smaller than the price per MB for main memory. The difference is even greater when compared to NVRAM.

It is the combination of these characteristics that has kept hard disks ahead of other types of storage such as RAM, NVRAM, tape, CDROM, DVD, jukeboxes, and floppy disks in the choice for on-line storage. For example, even though main memory has faster random access and tape is cheaper than a hard disk, main memory is also much more expensive than hard disk per MB and tape does not support random access. To increase the capacity, speed and/or reliability of disk storage, computers can also use multiple disks (e.g., a RAID system).

More than ten years ago, some people (from the 'disks-are-dead' camp; for example, see [Gray and Reuter, 1993]) predicted that in the near future hard disks would be replaced by faster and cheaper, high-density memory chips. To this day, however, this change has not happened. On the contrary, disks have continued to play an important role, and it seems that hard disks will remain the main hardware for storage systems in the foreseeable future.

1.2 Problem Overview

Since disks play a central role in storage and many applications are data intensive, the overall quality and performance of the entire storage system are strongly affected by them. In order to improve the quality and performance of disks, we do not want to change the disk itself, but want to focus on how the disks are used by applications. In simple terms, the disk is just the hardware that is used by software to store data of users.

Software programs that use disks come in different forms and sizes. Two very common types of software using disks are file systems and database management systems (DBMSs). A file system is usually part of an operating system, which is the encompassing software that lets all hardware components of a computer, such as the hard disk, the printer, the keyboard, the video adapter, etc., work together as one unit. The file system is the part that takes care of data storage, and usually supports data containers such as *files*

[†]In this dissertation, 1 GigaByte (GB) = 2^{30} bytes, 1 MegaByte (MB) = 2^{20} bytes, and 1 KiloByte (KB) = 2^{10} bytes, unless stated otherwise.

and *directories* to help users structure their data. A DBMS is a large and complex piece of software that is tailored to store large amounts of data, to maintain their integrity, and to manage (concurrent) access to those data. A DBMS can be implemented on top of an operating system, so that it uses the file system within the operating system to store its data, or it can be implemented to bypass the file system and to manage the data storage itself. The reason for bypassing the file system is often performance, as there is a mismatch between the services needed by a DBMS and the services offered by a general file system (see e.g., [Stonebraker, 1981]).

The quality and performance of data storage is influenced by how a file system or DBMS uses the disk. In this section, we identify three types of problems that relate to the current usage of disk.

(1) Data integrity problems: how to maintain the integrity of the data stored on disk, even under system failures and power failures.

(2) Performance problems: how to alleviate the I/O bottleneck.

(3) Modularization problems: how to strive for better modularity of the software using disks.

1.2.1 Data Integrity Problems

The first type of problem concerns the quality of the use of the functionality provided by disks. The main purpose of disks is to store a user's data efficiently and — in our opinion more importantly — reliably. The data stored on the disk are important to their owners, and therefore, their integrity must be maintained. Only in a few instances do the owners not care much for the integrity of their stored data. For instance, it is usually not a problem if an application that caches data, such as a web proxy, loses its data after crashing. However, in many other cases, the integrity of data is important to users. Unfortunately, often there are trade-offs between achieving good performance and maintaining data integrity.

For example, consider the simple act of overwriting a piece of data on disk, which is called an **in-place update**. A power failure during the overwriting of this piece of data could result in a situation where partly the new value and partly the old value of the piece of data is stored on disk. Such an outcome may turn out to be disastrous as *both* the new version as well as the old version, which may have been stored 'safely' on disk a long time ago, are then lost.

Another example that endangers the integrity of data stored on disks is the lack of support to execute multiple operations as one atomic action (**atomicity**). Many applications need to update multiple pieces of data on disk atomically, as all the updates together form one logically indivisible operation. For example, in a file system the logical operation of deleting a file usually consists of writing three updates to disk: an update to the directory of the deleted file, an update to the bitmap that keeps track of free blocks, and an update to the bitmap that keeps track of free i-nodes. An *i-node* is a data structure that is used by some file systems to hold meta-information of files, such as the owner of a file, and to keep track of the disk blocks that hold the contents of the file. A system failure may have the effect that not all these updates reach the disk, which yields an inconsistent file system, which, in turn, may result in loss of data.

Last, we mention the danger of command reordering for data integrity. **Command reordering** refers to the fact that the order in which users send write commands to the disk and/or receive acknowledgments from the disk may not be the order in which the results of said commands actually become persistent on disk. For example, disk scheduling and/or write caching in the disk may cause such a command reordering. Command reordering can yield substantial performance improvements, but it can also seriously complicate the recovery process after a crash. The complications arise because the order in which data are written to disk is often vital to the consistency of data on disk. Therefore, if users cannot control the order in which data are written to disk, the integrity of data on disk becomes difficult to safeguard.

The quality of a storage system is for a large part measured by the guarantees it provides its users with respect to the integrity of the data it stores. Since most storage systems use disks to hold their data, it is important they use disks in such a way that data integrity can be guaranteed. These storage systems can only provide such data integrity guarantees, if they are built on a sound base. An interface to the disk with operations that have clear semantics with respect to data integrity provides such a sound base.

1.2.2 Performance Problems

The second type of problem concerns the performance of disks. For the performance of a memory device two aspects are crucial: its access time and its bandwidth. The **access time** is the time needed to get access to the first byte of certain data to be read or written. The **bandwidth** is the speed at which the rest of said data can be read from or written to the memory device in question. In comparison with main memory, the average access time of a disk is large and the average bandwidth of a disk is small. As a matter of fact, disks are often named to be the limiting factor in the overall performance of a computer system, that is, they are the cause of the I/O-bottleneck [Ousterhout and Douglis, 1989; Ousterhout, 1990]. Even though advances in technology have improved the access times and the bandwidth of disks over the years, these improvements have not removed the I/O-bottleneck. After all, the speed of electronics, such as main memory and processors, has increased even more.

Consequently, since the disk forms a bottleneck, it is worthwhile to improve the performance of disks, as this will often result in an improvement of the overall performance of the computer. One performance problem of current disk usage is the low utilization of the available disk bandwidth. Disks are capable of transferring many MBs per second. Unfortunately, this bandwidth can only be achieved if the data are stored on disk in large consecutive ranges. If the disk is requested to read or write small amounts of data that are spread across the disk, the achieved bandwidth quickly drops to less than 5% of its maximum value.

Another problem with current disk usage that lowers disk performance is formed by the use of synchronous writes to uphold the integrity of the data on disk. A write request is synchronous if the application issuing the request (e.g., a file system or DBMS) waits until the write has completed before continuing. Applications use synchronous writes to force data to be written to disk in a certain order, which increases the likelihood that the integrity of the data on disk can be restored after a crash.

Unfortunately, the method of using synchronous writes to enforce a certain write order often yields low disk bandwidth usage and high latency. The reason for this performance loss is twofold. First, after issuing a synchronous write, an application sits idle for a relatively long period of time waiting for the synchronous write to complete. In contrast, after issuing an asynchronous write, the application could immediately continue to do something useful. Second, the enforced write order is not always the most efficient order to write the data to disk. Note that these performance-degrading synchronous writes were introduced to overcome another problem, namely the lack of support for writing multiple blocks to disk in one atomic action (see also Section 1.2.1).

To limit the impact on performance, many file systems only use synchronous writes to enforce the write order of metadata updates to disk because these metadata are vital to the consistency of the structure of a file system on disk. Nevertheless, even with this selective use of synchronous writes, the degradation in performance can be so severe that some file systems, such as the Ext2 file system [Ext2FS; Card et al., 1994; Beck et al., 1998], which is used in the operating system Linux, choose not to use synchronous writes at all, and are purposely willing to endanger the integrity of the data stored on disk for the sake of performance. Unfortunately, it is ironic that some operating systems (such as FreeBSD) that do use synchronous writes for integrity purposes turn on write caching for their disks by default, which can actually counteract the ability of synchronous writes to safeguard data integrity (see also Section 9.4.8).

1.2.3 Modularization Problems

The third type of problem concerns the complexity of the software using the disk. Every storage system, such as a DBMS or a file system, contains support for disk management. This disk management module is developed and implemented for each storage system almost from scratch again. Since this module more or less provides the same functionality in each storage system, it seems attractive to create one generic and reusable disk management layer. Development and implementation of a storage system can then become easier due to reusability of the generic disk management layer.

Such a disk management layer could take care of the low-level details of managing a disk, such as disk block allocation, suitable clustering, etc. The software layers on top of this disk management layer then only have to concentrate on implementing higher-level concepts, such as relational tables, files and directories, access rights, etc. Moreover, a separate disk management layer could provide an interface with clear semantics with respect to data integrity guarantees in order to make it easier to create higher-level storage systems that protect the data of users against loss and inconsistency.

1.3 Existing Systems

In the past, many techniques have been proposed to solve one or more of the problems mentioned in Section 1.2. In this section, we will give a brief overview of some existing systems and the techniques used in those systems to overcome some or all of the problems.

1.3.1 Journaling File Systems

A number of file systems use a technique called journaling, also referred to as write-ahead logging. This technique, widely used to guarantee data integrity, enables a group of changes to data on disk to be executed as an atomic unit. To achieve atomicity, a system using journaling first enters into a log the changes it is about to make to the data on disk. The log itself is stored in its own separate part of the disk, or on another physical disk. After the log has been safely written to disk, the system then performs the actual changes to the data on disk.

If a system failure occurs while the system was writing entries into the log, the original data on disk have not been changed yet and their integrity is therefore intact. If a system failure occurs after the log has been written, but while the system is making the changes to the data on disk, the log is used to restore the integrity of the data after the system comes up again. The log describes the changes the system was about to make, and the system can redo them so that the integrity of the data is restored.

Examples of journaling file systems are Ext3 [Tweedie, 2000], Episode [Chutani et al., 1992], Cedar [Gifford et al., 1988; Hagmann, 1987], ReiserFS [Reiser], XFS [Sweeney et al., 1996], and JFS [JFS; Best, 2000]. Unfortunately, for efficiency reasons, all but one of these file systems use journaling only to guarantee the integrity of metadata, that is, they only guarantee the integrity of the file system structure, not the user's data stored within the file system. Only Ext3 provides the option to safeguard a user's data as well. Just as DBMS engineers, however, we believe that not protecting the user's data is disregarding their importance.

1.3.2 Disk Scheduling

One technique widely used in disk systems to improve their performance is disk scheduling [Geist and Daniel, 1987; Seltzer et al., 1990; Teorey et al., 1972; Teorey, 1972; Worthington et al., 1994; Worthington, 1995; Lumb et al., 2000]. Disk scheduling involves reordering incoming disk requests in order to increase disk throughput (i.e., the number of requests a disk can handle per unit of time). Disks have mechanical moving parts and these parts must be positioned correctly before the disk can read or write a particular piece of data on the disk. Positioning takes a relatively large amount of time compared to the time it takes to actually read the requested data from or write the provided data to the disk. By reordering the requests, the disk can minimize the amount of positioning it has to do, and thereby minimize the time lost in positioning. This process is similar to the way a package delivery service chooses its route when delivering packages to homes across town. The less time is lost in positioning, the better the throughput of the disk.

Unfortunately, reordering disk requests endangers data integrity. If a user requests a storage system to write data items A, B, and C, in that order, the user may be very surprised if, after a crash, data items A and C are on disk, but data item B is not. Therefore, even though disk scheduling may improve performance considerably, it should be used with caution because it complicates recovery, and may even lead to loss of data.

1.3.3 Modularity

In the past, other general disk management layers have been proposed. Loge [English and Stepanov, 1992] and Mime [Chao et al., 1992] are two examples of storage systems that provide such a general disk management layer. Both systems also provide some performance enhancements to improve the utilization of the available disk bandwidth, and provide guarantees for user data integrity. Unfortunately, with these systems users cannot indicate which data they like to be clustered on disk. Clustering data on disk (e.g., storing data consecutively on disk) is important for the performance because disks are good at accessing data that are stored close to each other. Furthermore, both Loge and Mime require changes to the hardware of the disk, or at least modification of the disk's firmware.

Introducing a general disk management layer is only one way of improving the modularity of computer software. Recall that our aim is to put the low-level functionality (i.e., disk block management) of a storage system, such as a file system or DBMS, into a separate disk management layer. In the stackable file system design [Heidemann and Popek, 1994, 1995; Khalidi and Nelson, 1993] a file system is built up of more than two layers. The stackable file system design also puts high-level functionality, such as compression or remote access, in separate layers. Each layer adds new functionality to the file system. Such a design is more general than what we propose to do.

1.4 Logical Disk

An attractive approach to solving the data integrity problems, alleviating the performance problems, and improving the modularity of storage systems is the creation of a new software layer. This software layer is created between the disk hardware and the DBMS and/or file system software providing a new disk abstraction. The advantage of a new disk abstraction is that we can use existing hardware. In order to investigate whether this idea is viable, we designed and implemented a disk management software layer called **Logical Disk**, or **LD** for short. In the future, if LD's ideas have proven themselves successful, an integration into a disk controller may be considered.

The main goal of the LD project presented in this dissertation is the creation of a disk management software layer that improves the use of disk hardware. In particular, with LD we aim to:

(1) Improve the modularity of storage management software,

(2) Improve the functionality of disk storage subsystems by improving in particular their data integrity properties, and still

(3) Provide performance competitive to other systems.

The design of LD that we will present in this dissertation is a considerable improvement of an earlier design [de Jonge et al., 1993; Grimm et al., 1996]. The current design requires significantly less main memory, scales better, and has a more suitable interface. Furthermore, our current design is more log-based than log-structured. A more detailed

enumeration of the differences between the current design and the earlier design is given in Section 1.4.2.

1.4.1 Characteristics of the Logical Disk

Below we briefly discuss a number of characteristics of the current design, which enable LD to offer well-defined data integrity guarantees and still deliver good performance.

- *Logical addresses and address mapping*
  The unit of storage in LD is a logical block. The contents of logical blocks are stored on disk in physical disk blocks. LD provides applications access to logical blocks via logical block addresses, which are mapped internally onto physical addresses. Applications need not bother about the physical locations nor about a disk block address administration. The address mapping allows LD to change the physical locations of data on disk during normal operation without affecting the application (*location transparency*). For example, LD may change the locations of data on disk in order to improve their physical clustering.

- *Disk files and disk clusters*
  LD supports the abstractions of a disk file, which groups one or more logical blocks into a larger logical unit, and a disk cluster, which groups one or more disk files into a larger logical unit. Using disk files and disk clusters, applications can indicate a logical relation between logical blocks and between disk files. For example, all files within one file system (or one directory) can be stored in one disk cluster.

- *Automatic physical clustering*
  On request, LD will keep the physical blocks of a disk file physically clustered as good as possible (intrafile clustering), which improves sequential read performance. The actual task of clustering the blocks on disk is transparent to the application. The application itself must indicate which disk files should be physically clustered and which not. After all, the application knows in general best for which disk files physical clustering would be beneficial, and for which disk files the blocks will be accessed randomly. Clustering of the disk files within one disk cluster (i.e., interfile clustering) can also be requested in a similar way.

- *Collective writes*
  In order to improve write performance, LD will batch many small write requests and write all accumulated data together in one large sequential disk write. This technique avoids the positioning time that would have been necessary for each of the small write requests. Instead, the positioning time now only has to be paid once when the disk writes all data in one large write. Consequently, the available disk bandwidth is used more effectively; especially if the writes were to places widely spread apart across the disk (i.e., random writes).

- *Command streams*
  A command stream is a logical channel for applications to issue successive commands. LD guarantees that the commands issued in the same stream will be executed in such a way that it seems as if they have been executed in the order of

arrival. Therefore, a user will never be confronted with the undesired side-effects of disk scheduling. Internally, LD may make any optimizations it can, including disk scheduling, as long as the final result preserves these consistency semantics.

- *Atomic Recovery Units*
  Atomic multiblock writes are supported in the form of Atomic Recovery Units (ARUs). A user can order LD to treat multiple write requests as a single ARU. LD guarantees that after a crash LD will recover to a state in which either all or none of the write requests in the ARU have been executed. Furthermore, ARUs offer a limited form of isolation, as the effects of an ARU (e.g., block writes within the ARU) are not visible to others until the ARU has been committed.

- *No in-place updates*
  LD banishes in-place updates; in other words, LD never overwrites existing data, but always chooses unused disk space to write data to. Consequently, a power failure can never cause valid data to become inconsistent.

- *Correct and fast recovery*
  After a crash, LD will recover to a recent and consistent state. Because LD uses a log, which contains the latest changes to the data on disk, and checkpoints, recovery is fast and relatively simple.

- *Log-based*
  LD is log-based (or log-enhanced), but not log-structured. LD only uses a small part of the disk as a log to support atomic multiblock writes, consistent recovery, and good write performance. The lion's share of the disk, however, is used to store data in a way that allows the data to be clustered according to the application's wishes. In log-structured file systems, such as Sprite-LFS [Rosenblum and Ousterhout, 1991, 1992], the entire disk is used as one huge log.

- *Integrated log*
  LD's log is an integrated part of LD's storage structure; data that are written into LD's log are immediately accessible in the normal way for read operations. In contrast, many other systems using a log, use the log only for recovery purposes. In such systems, data are first written into the log and subsequently into the storage area where the system normally stores its user data. The log is read only during recovery after a system failure.

  Consequently, without an integrated log, the buffers in main memory can only be reused after the contents of those buffers have been written both into the log and into the storage area on disk. In contrast, with an integrated log, the buffers can be reused as soon as their contents have been written into the log.

- *Direct segments*
  In a normal log-based system, all data are written twice: first in the log, then to another location on disk where they are stored more permanently. The advantage of using the log is that data integrity can be guaranteed even in the case of system failures. The disadvantage is a possibly substantial loss in performance. Therefore,

to increase performance, LD introduces the technique of direct segments. With this technique LD can, under certain conditions, avoid writing data twice, and can instead write data directly to a new location on disk, while still guaranteeing data integrity even in case of system failures.

- *Differential techniques*
  When writing a block of a disk file that needs to be clustered due to read performance requirements, the address of that block may undergo two address changes at a write request; first when the block is written in the log, and later when it is relocated to a preferred location elsewhere on disk. LD's address mapping uses a differential technique, which enables LD to efficiently support these kinds of double address changes, and which thus prevents multiple updates of metadata blocks (which would lower performance unnecessarily).

- *Staccato write*
  LD uses a special technique, called the staccato write technique, for writing new versions of metadata blocks to disk. The staccato write technique prevents in-place updates and yet is much more efficient than any other, more traditional approach (with or without in-place updates) for writing metadata.

- *Support for small block size*
  Logical blocks can be as small as the smallest unit that the underlying disk hardware can read and write: a disk sector, which is usually 512 bytes. The block size influences the amount of wasted disk space due to internal fragmentation. Analysis of the results of a survey on file size distributions in file systems [Irlam, 1993] (see also Table 9.6, on page 247) revealed that with 8 KB and 4 KB blocks, the wasted disk space is about 28% and 12%, respectively. In contrast, with 512-byte blocks, the internal fragmentation drops to only 1%. To support small blocks successfully, LD uses *automatic physical clustering* (see above) to avoid performance loss which is often associated with using small blocks. Furthermore, LD uses *compression techniques* (see Chapter 6) to keep the disk administration that is necessary to keep track of all those blocks, small and manageable.

1.4.2 Differences with an Earlier Design of the Logical Disk

An earlier design of LD [de Jonge et al., 1993; Grimm et al., 1996] already included some of the ideas presented in this dissertation. For example, it already promoted the idea of introducing more modularity in storage systems by introducing a separate disk management layer. It also introduced the idea of the Atomic Recovery Unit (ARU) as a method to group multiple operations into one atomic operation. Last, it also supported grouping blocks into larger logical and physical units, called **block lists**.

LD's current design is based on the same starting points as LD's earlier design. However, except for these starting points, the current version of LD is the result of an almost complete redesign, which was necessary to solve some problems with the earlier design. Consequently, there are many differences between the current design of LD and its earlier design. The major differences are summarized below.

- *Scalable main-memory usage*
  One main weak spot in the earlier design of LD was that it had a large main-memory footprint. Per GB of disk storage, the earlier design of LD used at least 1.5 MB of main memory. Therefore, with the advent of disks with capacities of several hundreds of GBs, the main-memory usage of this design would become a problem. Even though, nowadays computers with 1 GB of main memory are common, it is undesirable to spend such a large portion of it to LD alone.

  The earlier design of LD uses so much memory because it needs to keep a number of data structures in-core at all times, such as its *block number map*, which maps logical block addresses to physical block addresses. The reason for having to keep these data structures in-core was performance. For example, due to the design of the block number map, sequential access to the blocks of a single block list would often result in random accesses to the block number map. In other words, there was no *locality of reference*. Therefore, since caching is not effective for data structures without any locality of reference, it was necessary to keep the entire block number map in main memory for performance reasons.

  The current design of LD requires much less main memory because it is designed to use on-disk data structures, such as LD's address mapping (see Chapter 6), which maps logical block addresses to physical block addresses. For example, the design of LD's address mapping is such that there is locality of reference (see also the item on *offset addressing* below), and therefore, LD's address mapping has become a cacheable data structure. The current design of LD only requires main memory to cache recently used blocks to increase the performance of accessing and updating LD's data structures. Consequently, the current design of LD does scale up to large disks much better.

- *Improved internal data structures for the block administration*
  The prototype of the earlier design of LD used singly linked lists to implement some key data structures, such as the block number map and the block lists. As a consequence, the deletion of a logical block or a block list may require an expensive search from the beginning of a singly linked list to find the predecessor of the deleted block or block list. To prevent the expensive predecessor search, the user may provide LD with a hint which block or block list is the predecessor. Unfortunately, this design puts the burden on the user. The current design of LD uses variants of B-link trees to store its block administration, and uses the nice properties of trees to make accessing and updating information in the block administration efficient.

- *Scalable recovery time*
  The current design of LD uses a different recovery technique than the earlier design. The current design uses a checkpoint and logging technique, whereas the earlier design used a technique that involved scanning the entire disk. Even though with a proper layout of the disk, scanning the entire disk can be done relatively efficiently, it will still take on the order of minutes to recover with current disks that are several hundreds of GBs large. In contrast, with the current design of LD, checkpoints

can be made efficiently (see Chapter 7), and recovery can be done in a matter of seconds, regardless of the size of the disk (see Chapter 9).

- *Offset addressing*
  Both the earlier and the current design of LD use logical block addresses to access blocks. However, whereas the logical block addresses in the earlier design of LD do not have a semantical meaning (i.e., logical addresses are simply numbers without any structure), the logical addresses in the current design are actually tuples of the form (cluster_id, diskfile_id, offset). In other words, a logical block address in the current design of LD indicates the position of the block (offset) within the disk file it belongs to, and it indicates to which disk cluster that disk file belongs.

  Furthermore, in the earlier design of LD, logical block addresses were assigned by LD; users did not have any influence on the logical address chosen, they could only call the function `NewBlock` which would allocate a logical block address and return it. In the current design of LD, however, LD supports a sparse logical address space. Users can choose which logical block addresses to use.

  One important advantage of offset addressing is that it makes LD's address mapping cacheable. Consequently, LD's address mapping can be an on-disk data structure (see also the item on *Scalable main-memory usage* previously). Another advantage of offset addressing is that a file system on top of LD does not have to keep any administration at all on the logical block addresses that belong to files in the file system. For example, a file system can use a disk file to implement a file in the file system, and address the blocks of that disk file using offset addressing. The file system does not have to store the logical block address of each single block in the file individually.

- *Scalable ARU implementation*
  The implementation of ARUs in the prototype of the earlier design of LD used in-core lists for each ARU to keep track of the changes that were made to blocks in an ARU [Grimm et al., 1996]. The use of in-core data structures meant that the implementation could not support large numbers of concurrent ARUs nor large ARUs as that would require large amounts of main memory. The current design of LD integrates the ARU administration with the normal block administration (i.e., LD's address mapping), which is implemented by on-disk data structures. Consequently, the current design of LD can support large numbers of concurrent ARUs and large ARUs.

- *Improved semantics of ARUs*
  Even though the general purpose of ARUs has remained the same from the earlier design to the current design of LD, there are some differences. First, in the earlier design, the allocation of new blocks and block lists could not be executed within an ARU. In the current design, block allocation, and the creation of a new disk cluster or disk file can be done within an ARU. Second, in the earlier design, running ARUs could not be aborted; in the current design, running ARUs can be aborted. One application of ARUs is to implement transactions on top of them. Therefore,

since transactions can be aborted, it makes their implementation on top of ARUs easier if ARUs can be aborted as well.

- *A design of the reorganizers*
  The earlier design of LD also mentioned the possibility of reorganizing data on disk in order to improve the clustering of data on disk. However, it did not present a design of the reorganizers. This dissertation presents a discussion of the tasks of the reorganizers that must be performed in order to improve the clustering of data on disk. Furthermore, it introduces the design of a new data layout on disk to facilitate clustering data on disk.

- *Log-based vs. log-structured design*
  The prototype of the earlier design of LD was log-structured, similar to Sprite-LFS [Rosenblum and Ousterhout, 1991, 1992]. In other words, the entire disk was perceived as an append-only log. The disk was divided into segments and data were always written in a new segment. The advantage of a log-structured design is that it offers good write performance, but its sequential read performance may suffer, in particular due to lack of good physical clustering. The current design of LD is not log-structured, but log-based, as stated in Section 1.4.1. With log-based, we mean that only a part of the disk is used as a log; the rest of the disk is used for storing data in such a way that data can be clustered for good read performance. Consequently, the current LD can profit from the log as it provides good write performance, and it can use clustering to achieve good sequential read performance as well.

1.5 Contributions

The main contribution of this dissertation is the presentation, discussion, and evaluation of the design and implementation of the storage system LD. The design covers all major aspects of disk block management. In more detail, we can distinguish the following subcontributions:

- We show that it is possible to provide improved data integrity while still providing performance competitive to other storage systems.
- We introduce a disk block interface that supports location transparency (i.e., the users do not know where data blocks are placed on disk), but also allows users to indicate which blocks should be stored clustered to improve read performance. The actual act of clustering data blocks is left to LD.
- We present the improved (see Section 1.4.2) *Atomic Recovery Unit* as the abstraction to make multiple data block updates atomic, which is the building block that applications can use to guarantee data integrity.
- We introduce the technique of *direct segments* to improve the performance of user data logging storage systems.
- We present a design that uses the *differential* and the *staccato write* techniques to improve the performance of metadata updates.

- We introduce the design of a new disk layout that is used by LD to achieve good read and write performance. This design includes algorithms for reorganizer and cleaner processes that maintain the block layout.
- We present the results of extensive performance measurements and compare the performance of LDFS, a file system on top of LD, to the performance of several other file systems. The prototype implementation of LD used in these measurements does not yet implement the full design, but deviates a little from the original design, mostly for ease of implementation (see Section 9.2.2).

1.6 Outline of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 analyzes problems of current disk usage in great detail, and presents some general solutions that may be used to overcome these problems. Chapter 3 presents LD, which combines these solutions into a single integrated system. This chapter presents the supported abstractions and the external interface of LD.

The following five chapters (Chapters 4 – 8) present a detailed discussion of the design and implementation of LD. Chapter 4 presents a brief overview of the architecture of LD. It introduces the major components of LD and the interactions between these components. The structure of the discussion of LD's internal design is split up according to the four major types of data that LD distinguishes: log data, metadata, checkpoint data, and user data. The following four chapters each focus on one of these types of data.

Chapter 5 discusses the purpose and design of the log. LD not only uses the log to provide data integrity, but also to improve write performance. Chapter 6 discusses the internal data structures of LD that form LD's metadata. These data structures keep track of the data stored on disk by the users of LD. Chapter 7 discusses the recovery process of LD. The main focus of this chapter is the checkpoint, which is basically a snapshot of the data on disk. LD periodically makes a checkpoint to make recovery fast. Chapter 8 discusses how LD stores a user's data on disk. It presents LD's novel disk block layout and the reorganizer and cleaner processes that are responsible for maintaining the disk block layout.

The last three chapters of this dissertation (Chapters 9 – 11) evaluate the design of LD. In Chapter 9, a prototype of LD and a file system on top of it are presented, and their performance is compared to the performance of other file systems. The purpose of this comparison is to verify whether LD can provide performance competitive to other storage systems. Chapter 10 presents an overview of related work. The most common techniques and systems are discussed and compared to LD. The final chapter in this dissertation, Chapter 11, presents a summary of the research described in this dissertation, some conclusions, and a brief discussion of possible future work.

Chapter 2

Problems and Proposed Solutions

In the previous chapter we identified three types of problems related to the current usage of disks:

(1) Data integrity problems: system failures can compromise the integrity of data on disk, which leads to loss of data.

(2) Performance problems: the I/O bottleneck, caused by inefficient use of the disk, leads to performance loss.

(3) Lack of adequate modularization: data storage is a complex task; modularization helps to tackle this complex task.

These problems are the focus of this chapter. We take a closer look at each of these problems and suggest solutions. The challenge is to solve all the problems in an integrated way. After analyzing a problem we sketch possible solutions, emphasizing solutions that can be easily integrated. The actual integration of these solutions is part of our LD design as described in Chapters 3 – 8. We start with a brief introduction into disk storage and file systems to present the terminology and the model used in the rest of this dissertation.

2.1 Model

We start this section by taking a look at the main characteristics of a hard disk. Many design decisions of our Logical Disk have been influenced by these characteristics. This introduction offers a simplified look at disks and presents in few words the main properties of disks and defines the terms that will be used in further sections.

The introduction to hard disks is followed by a discussion of the software context in which disk storage is used. The software directly using disk storage is mostly the operating system. Within an operating system, multiple components can be distinguished. The one that is most relevant to our discussion is the file system. We will explain where our Logical Disk fits within this given structure.

2.1.1 Introduction to Magnetic Hard Disks

A **hard disk**, or simply **disk**, is a rotating mechanical device capable of persistently storing large quantities of data. A hard disk mainly consists of a disk assembly, a head assembly, and some electronics. Figure 2.1 shows a simplified model of a hard disk. The disk assembly and the head assembly are both shown.

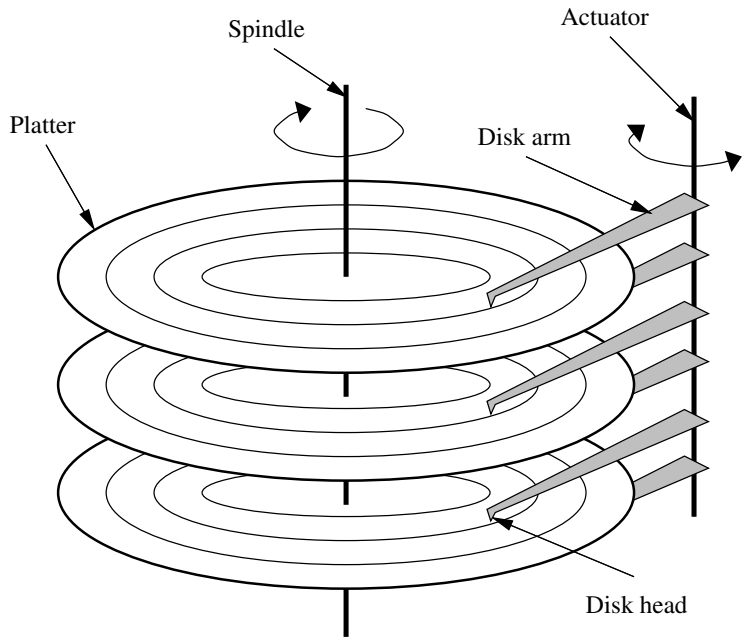


Figure 2.1: Simplistic view of a disk.

The **disk assembly** consists of platters and a spindle. A **platter** is a flat disk with a diameter of typically 3.5 inches that is coated on both sides with a layer of magnetic material. The platter itself is rigid, hence the name 'hard disk.' A disk can have multiple platters. For instance, hard disks intended for the PC-market typically contain one to five platters, largely dependent on the total storage capacity of the disk. High-end hard disks, used in large servers, can have up to a dozen platters. The platters are mounted, one above the other on a **spindle** that runs through the center of the platters. The spindle is attached to a motor, which makes the platters rotate at high speeds. The platters in current disks typically spin at 5400 to 10000 revolutions per minute.

The **head assembly** consists of disk heads, disk arms and the actuator. A **disk head** is a small device that can read and write magnetic patterns from the magnetic surface of the platters. Each platter has two heads assigned to it: one to read/write the top surface and the other to read/write the bottom surface. Each head is attached to a **disk arm** and the arms are all connected to one **actuator**. The actuator is a mechanical device that can

rotate about its axis and thereby move the arms which position the heads closer or farther away from the center of the platter. All arms are connected to the same actuator, so all the heads move in unison. In combination with the rotation of the platters this means that a head can reach every spot on the surface of its side of the platter. The heads never touch the surface of the platters. Instead they float on a very small cushion of air that is generated by the rotation of the platters. Only *one* of the heads can be reading or writing at a time.

Tracks, Cylinders and Sectors

Information (data) is stored in magnetic patterns on the surface of the platters. These patterns are organized in concentric circles on the platters. These circles are called **tracks**. The tracks closest to the spindle are called the **inner tracks**; the tracks near the edge of the platter are called the **outer tracks**. Tracks are numbered starting with the outermost track. A platter can hold tens of thousands of tracks. Figure 2.2 shows the surface of a platter with a number of tracks.

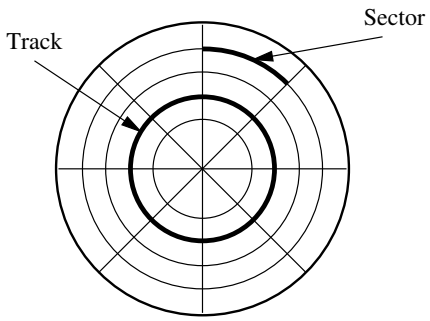


Figure 2.2: Tracks and sectors on a platter.

A track is further divided into **sectors**. A sector is the smallest amount of data that can be read from or written to a hard disk. A sector typically contains 512 bytes of data. In current drives, a track holds several hundred sectors. In the past, the sectors on a platter could be seen as pie slices, as depicted in Figure 2.2. In this model, each track has the same number of sectors. However, the physical length of a track gets longer the farther the track lies from the center of the platter. In other words, the sectors on the outer tracks are longer than the ones on the inner tracks. Consequently, this model makes inefficient use of the available surface area, since the outer tracks could hold more sectors. Therefore, modern hard disks use a technique called **zoned bit recording**, in which the number of sectors per track differs for inner and outer tracks. Each platter is organized into a number of **zones** (currently up to about 20 zones). Each zone consists of a number of consecutive tracks. Within each zone the number of sectors per track is constant. However, this number differs per zone: the farther away the zone is from the center, the more sectors there are per track. This way, the outer tracks have more sectors than the inner tracks, and

can therefore hold more data. Figure 2.3 shows an hypothetical example of a platter with two zones. The inner zone holds 8 sectors per track, the outer zone 12.

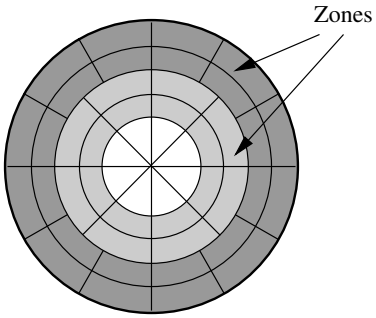


Figure 2.3: Platter with two zones: one with 8 sectors per track, the other with 12 sectors per track.

All tracks, across all platters, that are the same distance from the spindle in the center form a **cylinder**. Since all the heads move together as a single unit, the heads can read the information in the tracks of the same cylinder without the actuator having to move the position of the heads. However, in modern disks the tracks are so narrow and closely packed on the platters that it is very difficult for the disk manufacturer to exactly align the tracks of one cylinder one above the other. Expanding and shrinking of the platters due to temperature changes further complicate alignment. The deviations are minute, but since the tracks are so very thin, the heads must be precisely aligned with the tracks in order to read the data from the track correctly. Therefore, when another track on the same cylinder must be read, some movements of the heads may be necessary to align, or **settle**, the heads correctly over the track.

Head Switches, Cylinder Switches and Seeks

Sectors are numbered cylinder by cylinder, and within each cylinder track by track, starting at the outer cylinders and working their way inwards. This numbering determines the order in which the disk accesses the sectors when sequentially reading or writing an amount of data. Filling an empty cylinder with data starts at the first sector of the first track of the cylinder and continues with the following sectors. After all sectors of this track are filled, the disk continues filling the following track of the same cylinder. To fill this track the next disk head is activated, which is called a **head switch** and includes any settling that is needed to align the heads exactly over the track. When the last sector of the last track of the cylinder is reached, the disk continues with the first sector of the first track of the next cylinder. The switch required here is called a **cylinder switch**, which, confusingly, is also often referred to as a **track switch**. We prefer the term cylinder switch, because the term track switch is ambiguous. Both a cylinder switch and a head switch allow data from a different track to be accessed: a cylinder switch changes to the first track

on the next cylinder while a head switch changes to the next track on the same cylinder. Therefore, they can both be referred to as track switch.

The time it takes the disk to perform a head switch or a cylinder switch is called the **head switch time** or the **cylinder switch time**, respectively. We assume these values represent the time from the moment the disk stops reading/writing from one head up to the time it can continue its activity on the other head and includes the time needed to settle the head over the new track. A head switch takes typically at most 1 or 2 milliseconds. A cylinder switch takes in the order of a millisecond longer since it requires the disk heads to move to the next cylinder.

Moving the disk heads from one cylinder to another is called a **seek**. A cylinder switch is a special case of a seek in which the heads move to the adjacent cylinder: a one cylinder seek. The **seek time** is the time it takes for the actuator to move the heads physically from one track to another track including the time it takes for the heads to settle enough to begin reading the data that passes under the head. The seek time depends on the distance between the two tracks, but the relationship is not linear. If the seek distance doubles, the seek time will not be twice as long but will be a little less. The difference can be explained by the fact that moving the heads is a physical activity, which includes accelerating and decelerating the disk head. As a result, the speed of the disk head during a seek is not constant.

Head Skew and Cylinder Skew

The beginning and ending of each sector on the tracks of a cylinder are not exactly aligned above each other. Instead, the start of each sector in a track is a little offset compared to the same sector in the track below it, which is also located in the same cylinder. This offset is done to make the head switch as efficient as possible. Suppose that the sectors were not offset, but were all exactly aligned one above the other. As soon as the head reaches the end of the last sector on a track n, the next disk head also floats exactly over the end of last sector of the next track $n+1$. This last sector of track $n+1$ is immediately followed by the first sector of track $n+1$. However, activating the head that is able to access the sectors on track $n+1$ (the head switch) takes time, and in the meantime, the start of the first sector rotates past the disk head. Consequently, after a head switch one has to wait almost a full rotation before the first sector on track $n+1$ can be read.

Head switches are common, so this situation is not very good for performance. Therefore, disk manufacturers offset the start of this first sector a little in the opposite direction from which the platters rotate. This offset is called **head skew**. By the time the head switch has completed, the start of that sector has (ideally) just rotated underneath the head. In practice, the offset is a little larger to compensate for variations in the time it takes to settle the head over the track after a head switch.

Not only sectors within the same cylinder are offset, but also sectors of adjacent cylinders are offset from each other. This offset is done to compensate for the time lost in a cylinder switch. The offset between adjacent cylinders is called **cylinder skew**. The size of the offset is determined by the time it takes for the disk to go from the end of the last sector of the last track in a cylinder to the start of the first sector of the first track in the next cylinder. Since a cylinder switch not only requires a head switch but also requires

movement of the disk arm from one track to the next, a cylinder switch requires more time than a head switch.

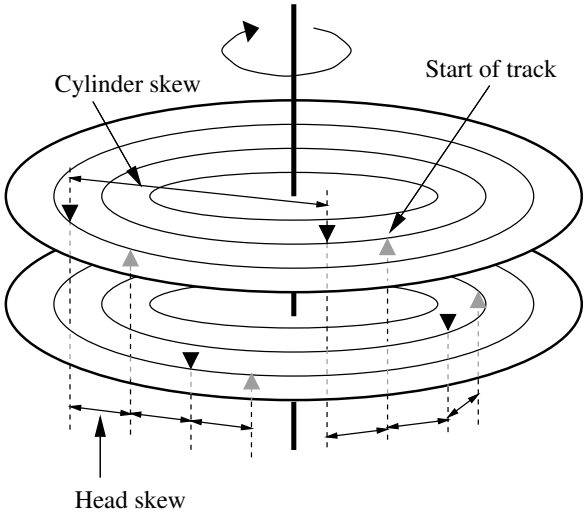


Figure 2.4: Head skew and cylinder skew.

Figure 2.4 illustrates the head and cylinder skew. The figure shows two platters, each with a few tracks. For simplicity, the platters do not have multiple zones. The start of the first sector of each track on both sides of the platters is indicated in the figure. The figure shows that the sectors of the tracks of one cylinder are not aligned one above the other. Each track is a little offset from the previous one: head skew. Looking at the first tracks of two adjacent cylinders, we can see a similar effect: cylinder skew.

Electronics

In addition to the mechanical components described above, the disk contains electronics to coordinate the proper working of the disk and head assemblies. It also takes care of the translation between data (bit streams) and magnetic patterns that the disk heads read and write on the surface of the platters. The translation also includes adding **Error Correcting Codes (ECC)** to the data. During reading and writing of the magnetic patterns on the platters errors can and frequently do occur. Most of these errors can be corrected using the redundant information that is stored within the ECC. The error detection and correction is transparent to the user of the disk. The 512 bytes that we mentioned earlier as the size of a sector is the net amount of data that can be stored in a sector by the user. Internally the sector is a little larger to hold the extra ECC information, as well as a preamble identifying the sector.

Every disk also contains some on-board memory. The capacity ranges from hundreds of kilobytes to several megabytes. This memory is used as a buffer that can be used for

read requests as well as for write requests. In the first case, data that has been recently read by the disk heads can be cached in this buffer. Future requests for the same data can then be served from memory, which is much faster than physically reading the requested data from the platters again. The buffer can also be used to hold data that has been read-ahead by the disk heads, in the hope that the data will be requested in the near future. If the buffer is used for writing, it could be used as part of a **write-back strategy**. In this case, the data of write requests are first stored in this buffer, and the process of physically storing the data in magnetic patterns on the platters is deferred to later. This strategy often improves the performance of disks, but also comes with risks. The buffer is nonvolatile memory, which means that data stored in the buffer is not persistent. If a power failure occurs, data in the buffer that has not been physically written to the disk yet, will be lost. How the memory is used and what caching algorithms are used is determined by the firmware in the disk's electronics, which is programmed by the disk manufacturer.

The disk interacts with the outside world via an interface. The two most common interfaces that are used are **ATA** (Advanced Technology Attachment), also known as **IDE** (Integrated Drive Electronics), and **SCSI** (Small Computer System Interface). Both interfaces have had many enhancements and the names of the interfaces have changed just as many times. Some of these new versions have names like Fast-ATA, Ultra-ATA or Ultra-Wide-SCSI. These interfaces define a set of commands that can be sent to the hard disk. The commands that we are interested in are the commands that order the disk to transfer data from or to disk: read and write commands. A read command typically instructs the hard disk to read a certain amount of consecutive sectors from a specific location on disk, and a write command instructs the disk to store a certain amount of data in consecutive sectors on a specific location on disk.

Putting It All Together

We have now discussed all major parts of a hard disk. What happens if the disk receives a read request for data in a particular sector? For sake of simplicity, we only concentrate on the mechanical parts of the disk, and leave the role that the electronics and buffer management play out of the discussion. The requested sector is located on a particular track within a particular cylinder. In order to read this sector, three actions must be done. First, the correct disk head must be activated. Second, a seek is done to position the disk heads over the correct cylinder. Third, the head must wait until the platter has rotated far enough that the requested sector is underneath the head. This delay is called the **rotational delay** or **rotational latency**. Only then the head is ready to start reading the requested data that is in that sector and to transfer it to the user that requested it. The time it takes for the disk to activate and position the head and wait until the correct sector has rotated under the head is called the **positioning delay**. Since the activation of the head and the positioning of the head can mostly be done in parallel, we define the positioning delay as the sum of only the seek time and the rotational delay.

2.1.2 Software Structure

The hard disk is the medium on which data are physically stored. In this section we look at the structure of the software that uses this hardware. Figure 2.5 shows a simplified view

of a computer system, which is built up of hardware and software layers. At the bottom is the actual hardware, like the hard disk, tape drives, CPU, memory, keyboards, monitors, etc. On top of this hardware runs the operating system, such as Solaris, FreeBSD, Linux, or WindowsNT. Its job is to control the hardware devices of the computer and let them work together. The operating system offers a clean interface of system calls that can be used by application programmers to implement their programs. These user application programs form the next layer. The kind of application programs can range from simple editors and email-readers to full-blown database management systems.

We will often use the term *users*. In the above context we used it to refer to humans: the users of an application program. However, we will also sometimes use the word in a more general sense. A user can also be a piece of software. For example, an application program is a user of the operating system, since it uses the functionality offered by the operating system. However, for clarity, we will use the word *client* as much as possible if we refer to pieces of software that uses some functionality offered by another piece of software.

Conceptually, the operating system itself consists of many components. Immediately above the hardware run the **device drivers**. Each device driver is directly responsible for controlling one type of hardware. The device driver for a hard disk (i.e., a disk driver) controls the underlying hard disk using the IDE or SCSI protocol.

The operating system also supports other concepts, such as **virtual memory**, **file system**, **threads**, etc. In the figure we have drawn these higher-level concepts as a sublayer above the device drivers. The file system is responsible for the persistent storage and manipulation of data. In this work we are mainly interested in the storage of data on hard disks, and we will therefore only concentrate on the file system and the disk driver of the operating system. In following figures, we will only show these parts of the operating system to avoid crowding the figure with irrelevant information.

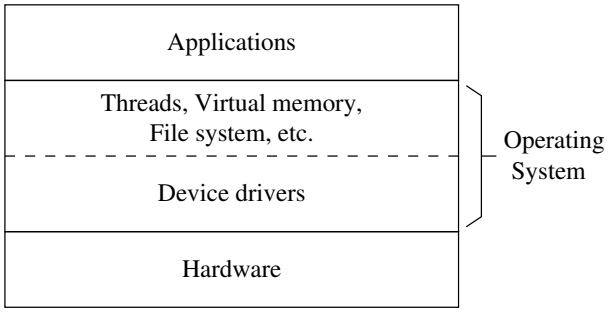


Figure 2.5: Software structure on top of the disk.

The file system offers its users a structured way to store data persistently. For example, in the UNIX world the file system provides the users with **files** and **directories**. Files are the basic container for data. The data within a file is simply stored as a stream of bytes, without any internal structure. Files are organized in a hierarchical structure of directories.

The file system also maintains a number of **file attributes** for each file. The number and type of attributes differs per operating system. Common examples of such attributes are:

- File size
- Owner information — who controls the file?
- Permissions — who is allowed to access and/or modify the file's contents?
- Timestamps — when was the file created, last accessed and/or modified?

One well-known storage application is a **database management system** or **DBMS**. DBMSs often require large amounts of storage to store data of users. Performance is often a very important aspect. However, the support that typical operating systems offer is not what a DBMS needs [Stonebraker, 1981]. Therefore, some DBMSs implement their own functionality and somehow bypass the operating system and in particular the file system. In that case, DBMSs do not use the file and directory support of the file system, but implement their own storage structure on the disk for the database tables and indices. Figure 2.6 shows the software structure of a DBMS bypassing the file system.

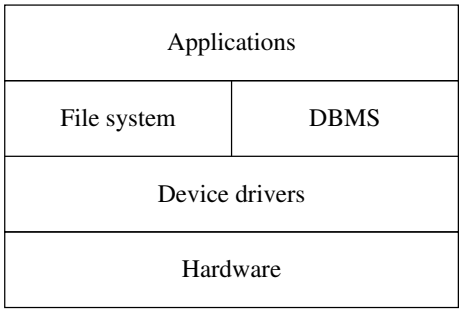


Figure 2.6: DBMS bypasses the file system.

2.2 Modularity

The first type of problem we look at is the lack of modularity in current file systems. File systems are complex pieces of software because they translate high-level concepts, such as files and directories, into low-level concepts, such as disk blocks. We see the file system as a composition of two clearly distinguishable subtasks. We call the task of managing the high-level concepts **high-level file management** or just **file management** and the task of managing the low-level concepts **low-level file management** or **disk block management**. The division of the file system in these two subtasks is depicted in Figure 2.7.

File management concerns the administration and maintenance of the high-level concepts of files and directories. For example, creation and deletion of files and directories,

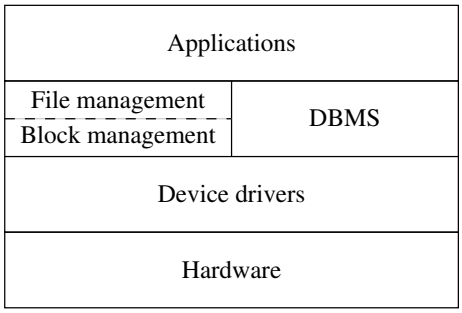


Figure 2.7: File management and disk block management.

and authorization issues are file management tasks. Disk block management concerns the physical aspects of storing those files and directories in disk blocks on the hard disk. These aspects include free space management and disk block allocation on disk.

Current file systems integrate both these tasks into one component. However, we strongly support the principle of 'separation of concerns' and therefore believe that these two tasks should be separated into two distinct layers. In addition, we also feel that the disk block management layer could be extended with new ideas to solve some other existing problems, which are described in the following sections of this chapter. In short, we propose to develop a new software layer that takes care of disk block management and more: the Logical Disk (LD). Such an extended disk block management layer would implement features that lighten the task of the file management layer. A file system built on top of LD could solely concentrate on file management. This approach would certainly reduce the size of the current file system component in operating systems at the expense of adding a new disk management component.

The advantages of a more modular approach are well-known from the field of software engineering. Modularity provides lower software maintenance costs, better reliability, re-usability of the disk management code, and cheaper and easier development of new file systems. LD as a separate module could not only be used by a file system, but a DBMS could equally well benefit from the added functionality. Since a current DBMS also contain lots of code dealing with low-level storage of data on disk, LD could reduce the complexity of a DBMS as well. Figure 2.8 illustrates this situation.

2.2.1 Location Transparency

What will such a new separate disk management layer look like? The interface exported by the disk management layer should hide the details concerning physical disk block allocation, block administration and other lower-level details, such as physical disk block addresses. A physical disk block address maps directly onto one or more consecutive sectors on disk. We want to hide such details from the higher levels. Therefore, one property that we want in the interface of the disk management layer is **location transparency** for data blocks. Location transparency has two main advantages. First, it shields the clients from the details of physically storing blocks on disk. Second, it opens the opportunity

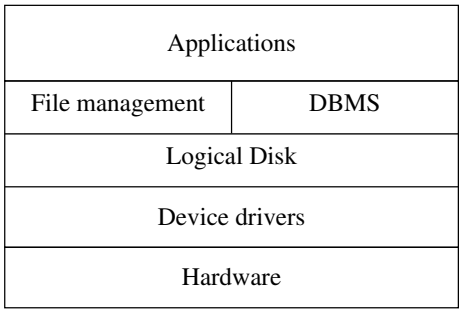


Figure 2.8: File management and a DBMS on top of LD.

for the disk management layer to move the blocks around on disk dynamically without bothering the file system. For example, it may be necessary to update the current layout of the blocks on disk to improve performance. This situation can arise if **aging** (see Section 2.4.2) has fragmented the disk so much that performance starts to suffer.

2.2.2 Clustering

However, denying the higher layers direct influence over the actual physical location of their data blocks also has a major drawback. The performance of a disk largely depends on an efficient placement of the blocks. Disk blocks that are often accessed together should, in principle, be stored closely together (i.e., clustered) on disk for best performance. The reason for clustering these blocks is that disks are efficient at accessing blocks that are stored closely together on disk. The performance of accessing blocks that are spread randomly over the disk is much worse. For example, blocks that belong to the same file are likely to be accessed together and should therefore be stored closely together on disk. In Section 2.4 we will look at this issue in more detail.

Normally, a file system knows which blocks are likely to be accessed together. It uses this information in its allocation algorithm to decide where to place blocks on disk in order to cluster the blocks for best performance. However, by introducing location transparency we take away that ability since a file system has no control over the actual physical locations of the blocks on disk anymore.

Fortunately, it is not necessary that the file management layer has full control over the exact locations of those blocks on disk in order to cluster the blocks on disk. The main point is that for performance reasons those blocks must be stored closely together on disk; the exact locations are of lesser importance. The disk management layer can take care of the actual placement of the blocks, if the file management layer could somehow specify which blocks should be clustered. Therefore, we must include a **data block clustering mechanism** in the interface of the disk management layer that can be used to indicate which blocks belong together and should be stored as a cluster on disk for performance reasons. It is the disk management layer's task to pick the appropriate locations on disk for those blocks.

2.2.3 Read-Ahead

Read-ahead is another enhancement that file systems use to increase performance. Based on current and/or past access patterns and knowledge of the physical layout of blocks on disk, the file system can decide to read ahead one or more blocks in the expectation that those blocks will also be requested shortly. Unfortunately, using logical block addresses instead of physical block addresses makes effective read-ahead harder to do because the information about the physical layout of blocks on disk is not known to the clients of our disk management system. Therefore, they cannot estimate how much effort it would take to read ahead extra blocks as they do not know where these blocks are located on disk. Retrieving these blocks would require a lot of effort if they are not in the vicinity of the current location of the disk head. Even worse, the effort may have been in vain, if it turns out that these blocks are not actually needed. Therefore, it is worthwhile to take the actual read-ahead costs into account when deciding which blocks to read ahead.

Higher layers often know when a read-ahead would be beneficial, but only the disk management layer knows the actual locations of the blocks and can make the final decision whether it is worth the effort to read the blocks ahead or not. Therefore, the interface of our disk management layer should also incorporate a mechanism that allows the file management layer to indicate when a read-ahead would be beneficial. The read-ahead should ideally specify how many blocks should preferably be read ahead and how much effort it may take. The disk management layer can then decide how much of this request it will grant, depending on the physical locations of the blocks on disk. Of course, in some cases, the disk management layer itself can also make educated guesses when a read-ahead would be beneficial, and act independently of the file management layer.

At first sight, it appears that by separating file management and disk management we only create more problems. However, we believe we can solve these problems in a sufficiently elegant and efficient way to make the gains of modularization outweigh the efforts required.

2.3 Data Integrity

The second kind of problem we have identified is data integrity. Data integrity is a very important aspect of any data storage system. Without it, clients cannot rely on the correctness of the data stored in the system. Maintaining data integrity is especially important in case the system crashes. Such failures can have different causes. Two types of failures can be distinguished:

- **System failures**
  These happen when the system crashes due to a power failure, software or hardware errors. Typically the system stops and the contents of volatile memory (RAM) are lost. Data stored on persistent storage media, however, remain intact.

- **Media failures**
  These happen when disk hardware fails. For example, a hard disk can physically stop to function due to head crashes or failing disk circuitry. The integrity of data stored on the storage media may be compromised.

System failures are far more common than media failures. A media failure is especially dangerous because it involves storage media failures. If the storage device, such as a disk, fails, then all data stored on it may be lost. Even though data on hard disk are supposed to be stored persistently, they can still be lost in such an event. The only possible way to reliably recover the data may be to restore the data from a back-up medium, such as a tape-archive or another disk.

In this dissertation, we only focus on maintaining data integrity in the face of system failures. System failures can lead to loss of data integrity because they can happen during the process of writing data to the storage medium. Furthermore, the contents of volatile memory are usually lost after a system failure, which may also lead to loss of data integrity because volatile memory is often used to buffer data before they are actually stored on the storage medium. From now on, we will use the terms 'system failure' and 'system crash', or simply 'crash', interchangeably.

In the next paragraphs we identify a number of specific causes that may lead to data integrity problems. Most, if not all, of these causes exist in current systems, and have to be dealt with in order to guarantee data integrity. For these specific causes, we present solutions that will guarantee data integrity even in the face of system failures.

We present these solutions within the framework of LD, our separate disk management layer. We feel that data integrity is such a basic requirement that should be shared by all clients, that it is logical to implement it at a very low level. Our disk management layer seems a right place for it. Every file system or other application built on top of it can then immediately benefit from the improved data integrity.

2.3.1 In-Place Updates

As a first cause of a data integrity problem, consider the simple act of overwriting a disk block: an **in-place update** of a single block. A power failure during the writing of a disk block may result in a disk block partly containing the new value and partly the old value. Losing both the new version and the old version (which may have been stored 'safely' on disk a long time ago) may turn out to be disastrous. Jim Gray has once rightly referred to in-place updates as 'poisoned apples' [Gray, 1981].

The Error Correcting Codes that hard disks write in each sector cannot solve the problems of in-place updates. The ECC codes are only useful to detect and correct small errors that occur during the read and write process. It cannot correct all errors. If a sector is only partly overwritten, the ECC code enables the hard disk to recognize that the sector has been incompletely written, but it is unlikely that it can correct the error. The disk can only return a read error when it receives a read request for such a sector.

Even if hard disks can write sectors atomically, the threat of data loss still exists. Most file systems do not use the sector as the unit of data transfer. It is very common for file systems to read and write data in *blocks*, which consist of multiple consecutive sectors. Blocks of 8 KB are not uncommon. If a power failure occurs during the in-place update of such a block, some sectors of the block will be correctly overwritten, while others may still hold the old value, leaving the block in an inconsistent state.

There are two straightforward solutions to prevent loss of data caused by in-place updates. In both solutions, the main idea is simply not to directly overwrite data. The first

solution is to first safeguard the old version of such a block by copying it to a new location on disk before overwriting the old version on the original location with the new version. The other solution is to write the new version to a new location and thus avoid the direct in-place update.

In the latter case it may be subsequently required to move the new version to the location of the old version, that is, to overwrite the old version with (a copy of) the new version. In particular, this may be required for keeping or getting a desired degree of physical block clustering, which would result in good performance for sequential read requests. Even though this copy is actually an in-place update, it does not pose a threat to data integrity, because the new version is already safe on disk. In the event of a system failure during the in-place update the new version can always be recovered from disk.

The movement of the new version to the location of the old version (or any other preferred location) can often be postponed to a more convenient moment in time. For example, it can be postponed to a time when the system is idle, which means that during busy times the system will not be overloaded. Being able to postpone the work to quieter times means that the workload can be spread over a longer time, which is good for the responsiveness of the system. It may even turn out that moving the new version to another location is not necessary at all, for example, because in the meantime the block has been deleted or has been overwritten again. In contrast, the solution mentioned first always requires two disk operations, as it must always first copy the old version of the block before it can overwrite it with the new version. Therefore, the second solution is clearly more efficient than the first one.

2.3.2 Disk Scheduling

The mechanical components of a disk are also its bottlenecks (see Section 2.4.1). In particular, the seeks and rotational delays are two very important factors for the performance of disks. Disk performance can often be improved considerably by reordering read and write requests such that disk-arm movement is minimized [Geist and Daniel, 1987; Seltzer et al., 1990; Teorey et al., 1972; Worthington et al., 1994; Worthington, 1995; Lumb et al., 2000]. However, the use of such **disk scheduling** techniques usually severely complicates the recovery of data after a crash. The main problem is that disk scheduling algorithms change the execution order of disk write requests. Consequently, the successful completion of a write command before a crash no longer implies the completion of all preceding write commands. In short, for the performance increase resulting from disk scheduling, a heavy toll must be paid in the form of considerably increased complexity of the recovery software required. The challenge here is to get an as good as possible performance while at the same time shielding the client from any (extra) burden to guarantee data integrity.

Disk scheduling can be implemented on different levels. It can be done in the file system, the disk driver or even in the disk hardware itself. Most operating systems incorporate disk drivers that do some kind of disk scheduling. Therefore, the file systems using those drivers must deal with the problems that disk scheduling brings with it.

One well-known method for avoiding some of the problems that result from disk scheduling is the selective use of **synchronous writes**. With a synchronous write the file system waits until the disk has written the data to disk, before it continues with the

next write. That way it can force the disk to write things in a particular order. In effect, it stops the disk driver from performing disk scheduling in a very cumbersome way. Unfortunately, synchronous writes are notorious for their negative effect on performance (see also Section 2.4.3). Therefore, their use more or less cancels the positive performance effects of disk scheduling. In fact, two synchronous writes are generally much worse for performance than two write commands of which the order may not be changed by the disk. This is why file systems limit the use of synchronous writes and only use them when writing system-critical metadata to disk for which data integrity is of great importance.

To avoid the risks of corrupting data integrity due to command reordering, several other solutions are possible. The first is to not use disk scheduling. All commands are executed in the order that they are given. The file system then has full control over the execution order. However, disk scheduling does have a noticeable positive effect on performance, so another solution would be preferable.

The second solution is that disk scheduling is allowed, but, after a crash, it must be possible to undo the side-effects of disk scheduling and restore the data on disk to a state that would have resulted if the original command order had been respected. This solution requires extra information about the original order and the reorderings done of all disk requests to be kept in some persistent way (i.e., probably some sort of log on disk). This information would then be used to aid the recovery process, which runs after the system restarts and brings the data to a consistent state again.

However, we prefer another solution. Note that the order in which two commands are executed is not always relevant as some commands may be independent of each other. For example, suppose that two users, independently of each other, create a file in different directories. For consistency reasons it is not important which file is created first in this situation, as long as each creation is itself atomic.

In general, a separate disk management layer will have no knowledge about which commands are independent and which are not. It can therefore not safely reorder commands to increase performance. Therefore, we choose to let the file management layer supply the disk management layer with this knowledge. Basically, this information specifies the desired partial ordering of the commands to the disk management layer. To this end, the disk management interface should provide functionality for users to indicate essential ordering requirements. This way, the disk management layer can reschedule some of the commands to improve performance, but also take care that it does not reorder commands whose order has been especially requested by the file system. The advantage of this scheme is that it gives the client the freedom to choose which operations to execute in order and also allows him to specify when the order is irrelevant so that the disk management layer can choose the execution order to optimize performance. How this is implemented in our disk management layer LD will be discussed in the following chapters.

2.3.3 Lack of Atomic Multiblock Writes

The two previous subsections illustrate that to obtain consistency, it is important to provide disk operations with the right semantics. Another essential semantic property is the ability to group multiple write operations into one atomic action. Many applications would benefit from this property. For example, in a UNIX file system the creation of a new file

requires updating both an i-node and a directory entry. The use of an atomic multiblock write operation for updating both an i-node block and a directory block, would guarantee recovery to a consistent state after a crash. Specifically, this would guarantee that we are left with either both old values or both new values, but never some mix.

This particular example refers to the need for a multiblock write operation that is guaranteed to be an atomic unit of recovery in order to ease the correct maintenance of a file system's metadata. Similarly, there is also a need for better support of atomic updates to the file-system's client data.

It seems logical to support these atomic multiblock writes at a low level, such as our disk management layer. We will support atomic multiblock writes in our disk management layer in the form of **Atomic Recovery Units (ARUs)**. An ARU is a group of write commands that will be executed as one atomic action. ARUs are useful in cases where multiple updates, such as disk block writes, are required to bring the system from one consistent state to another. Without ARUs as basic building blocks to guarantee atomic state transitions, it would require much more effort from clients to obtain data integrity, especially in the face of system failures.

2.3.4 Block-Level Transaction Support

ARUs are not the same as **transactions** [Gray, 1981; Gray and Reuter, 1993; Bernstein et al., 1987]. Transactions are more general and have three characteristic properties: *atomicity*, *isolation*, and *durability*. Atomicity refers to the all-or-nothing property of the operations within a transaction. Isolation deals with the correct execution of concurrent transactions. In particular, a crucial aspect is ensuring that the execution order of these transactions is **serializable**. In short, this means that the effects of this execution order are the same as if the transactions had been executed one after the other, which is called a **serial schedule**. If the execution order is not serializable, unexpected and incorrect results may occur, which violates the integrity of the data. Concurrency is important for performance in multiuser/multitasking systems, so in such systems isolation is a necessary property to maintain data integrity. The last property is durability, which guarantees that the operations within a transaction are recoverable after the transaction has been committed. Of these three properties, only atomicity is needed to implement atomic multiblock writes. Atomicity is also relatively easy to implement, therefore in the first design of our disk management system we limited the ARU to this property only.

In literature, transactions are commonly associated with the *four* **ACID**-properties: atomicity, consistency, isolation, and durability. The *consistency* property states that the complete execution of a transaction should take the system from one consistent state to another consistent state. However, this is usually considered the responsibility of the programmer that uses transactions. The correct execution of a transaction should not leave the system in an inconsistent state. Since this property has more to do with correct usage of a transaction than with the properties of the transaction that are guaranteed by the system, we have left it out in our discussion.

The logical next step in the development of our disk management layer is to extend Atomic Recovery Units to full **block-level transactions**. Specifically, this means supporting the three transaction properties *atomicity*, *isolation* and *durability*. Usually the

responsibility for transaction support is placed in higher software layers. Nevertheless, it may be wise (see also Section 2.2) to already support transactions at the disk management level. In that case we only have to solve the problem of transactions once, and all higher layers can then benefit from the extra functionality. Otherwise, every application has to implement (parts of) the functionality itself. Offering full transaction semantics to all higher software levels would become most attractive if it is possible to implement block-level transactions in such a way that applications not needing them are not adversely affected. We believe that it will be possible to achieve this, however, full transaction support is not in our current design of LD.

2.4 Performance

In Section 2.2 we already identified two performance-related mechanisms that should be included into the interface of modular disk management software: data block clustering hints and read-ahead support. In Section 2.3 we discussed how to allow disk scheduling for improved performance while shielding the client from possible data integrity problems. In this section, we look at two other types of performance problems: low disk bandwidth utilization and slow synchronous writes. We will look at these two problems more closely and suggest solutions to improve performance without compromising data integrity.

We start by looking at the low utilization of the available disk I/O bandwidth when serving small read and write requests, which is one cause that is responsible for the well-known I/O bottleneck. This utilization can often be as low as 5%. This inefficiency is caused by the overhead inherent in mechanical devices such as hard disks. Where does this overhead come from and how large is it? In the next section we first demonstrate the overhead by means of a fictitious disk. After that we present a possible solution to this problem. Last, we focus on the second problem: synchronous writes.

2.4.1 Overhead of Seek and Rotational Delay

In principle, disks are capable of transferring data at high speeds of many MB per second (currently around 30 MB per second). However, that throughput can only be achieved if data is read or written sequentially from the disk. The overhead of **positioning delays** (seeks and rotational delays) often reduces the effective bandwidth of the disk to only a tiny fraction of its maximum.

Given the specifications of a disk, we can calculate its maximum bandwidth and analyze how the effective bandwidth utilization changes when transferring different amounts of data. Let us take a more detailed look at a fictitious (but typical) hard disk. The specifications of our fictitious current disk, which we will refer to as *disk A*, are given in the first two columns of Table 2.1. *Disks B* and *C* in the same table refer to more advanced disks which we will discuss later on.

We have given only one value for the number of **sectors per track**. However, disks usually divide the surface of a platter into multiple zones and each zone packs a different number of sectors per track, as was explained in Section 2.1.1. For simplicity, we assume

Table 2.1: Specifications of a fictitious current (Disk A) and two fictitious future disks (Disks B and C).

Property	**Disk A**	**Disk B**	**Disk C**
#Platters	4	4	4
#Heads	8	8	8
#Cylinders	15,000	30,000	30,000
Avg. #sectors per track	400	800	800
#Bytes per sector	512	512	512
Total capacity (bytes)	2.46×10^{10}	9.83×10^{10}	9.83×10^{10}
Spindle speed (RPM)	10,000	20,000	15,000
Head switch time (ms)	1.0	0.5	0.8
Cyl. switch time (ms)	1.5	0.75	1.2
Avg. seek time (ms)	5.0	2.5	4.0
Sust. transfer rate (MB/s)	27.7	110.6	80.5

that the value mentioned in the table is the average number of sectors per track (i.e., the weighted average over all the zones).

Given these numbers, we can calculate the **average sustained transfer rate (STR)** of a disk, which is the maximum speed at which that disk can read data over longer periods of time. More precisely, it is the average transfer rate achieved when reading the whole disk in the most efficient (i.e., sequential) order. Therefore, this number includes the overhead of the head and cylinder switches incurred when transferring large amounts of data. This overhead is larger for inner zones than for outer zones.

The average sustained transfer rate can be computed by first determining how long it takes to transfer one cylinder's worth of data. A cylinder consists of a number of tracks, which is equal to the number of heads. The size of a track is determined by the (average) number of sectors per track, where each sector is 512 bytes. Therefore, the size of a cylinder is found by multiplying the number of heads, the number of sectors per track and the size of a sector.

Now we need to calculate the time needed to transfer this amount of data. To transfer one track's worth of data, the head must wait for the platter to spin one whole revolution. The number of seconds that one revolution takes is 60 sec divided by the spindle speed, which is denoted as the number of revolutions per minute (RPM). The number of tracks we need to transfer a whole cylinder's worth of data is equal to the number of heads. We call this time the *pure cylinder transfer time*.

The word 'pure' already suggests that some extra time is added. We have called it *overhead*, and it consists of head and cylinder switches. After each full round, the next track must be read by another head, which requires a head switch. At the end, after the last track of the cylinder has been read, a cylinder switch is necessary, so that the disk can continue reading the first track of the next cylinder. Therefore, the number of head switches required to read one cylinder is equal to the number of heads minus one. The time needed to do the head switches and the one cylinder switch form the overhead for each transfer of a cylinder. The total transfer time needed to transfer the data is the pure

cylinder transfer time plus this overhead.

This calculation for the average sustained transfer rate can be expressed in a formula as follows:

$$STR = \frac{A}{B+C}$$

where

$$\begin{aligned}
A &= \text{avg. cylinder size in bytes} \\
&= \text{avg. \#sectors per track} \times \text{\#heads} \times \\
&\quad \text{\#bytes per sector} \\
B &= \text{pure cylinder transfer time in seconds} \\
&= \text{\#heads} \times \left(\frac{60}{\text{spindle speed}}\right) \\
C &= \text{overhead in seconds} \\
&= \frac{\text{cylinder switch time} + (\text{\#heads} - 1) \times \text{head switch time}}{1000}
\end{aligned}$$

For disk A the formula yields an average sustained transfer rate of approximately 27.7 MB/s. This transfer rate is the maximum bandwidth that disk A can sustain over longer periods of time during sequential access.

However, under normal use, the effective utilization of the bandwidth is much lower. Typically smaller and larger amounts of data will be read from and written to disk on different locations. The overhead of positioning the disk head over the correct location on disk will lower the effective throughput of the disk considerably. We can derive the actual bandwidth utilization when transferring x bytes of data from or to some random location on disk from the average sustained transfer rate number and the positioning overhead. The positioning overhead consists of two components: the seek time and the rotational delay. The average seek time for our disk A is 5.0 ms. The rotational delay is on average half a rotation of the disk which takes 3.0 ms for a disk spinning at 10,000 RPMs. The total delay is therefore 8.0 ms, before the actual data transfer can begin. The utilization of the bandwidth can be calculated as:

$$\begin{aligned}
Utilization &= \frac{\text{time to transfer } x \text{ bytes}}{\text{time to transfer } x \text{ bytes} + \text{positioning overhead}} \times 100\% \\
&= \frac{x/STR}{(x/STR) + \text{overhead}} \times 100\% \\
&= \frac{x}{x + \text{overhead} \times STR} \times 100\%
\end{aligned}$$

where

$$\begin{aligned}
STR &= \text{avg. sustained transfer rate (in bytes/second)} \\
x &= \text{amount of data (in bytes)} \\
\text{overhead} &= \text{avg. positioning delay in seconds}
\end{aligned}$$

The utilization can be depicted as a graph and is shown in Figure 2.9. The figure shows the utilization graphs of all three disks specified in Table 2.1.

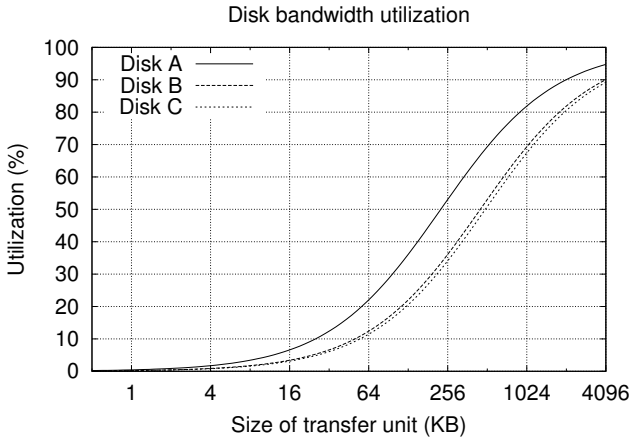


Figure 2.9: Disk bandwidth utilization.

The graph for disk A shows that the average transfer size must be more than 227 KB in order to get a utilization of at least 50%. From this figure we can conclude that the overhead of seek and rotational delay is quite substantial for small amounts of data, and that, as a consequence, the bandwidth of a current disk can only be used effectively if data is transferred in large sequential amounts. In other words, it is a worthwhile approach to try to restrict disk I/O to large sequential transfers as much as possible, as this may considerably improve read and write performance.

Similar calculations can be made for any hard disk configuration. In the future, improvements in technology may lead to faster disks and the utilization graphs will change accordingly. Table 2.1 also lists the specifications of two possible future disks. Assuming that spindle speed, disk-arm speeds, and data density will all double in the near future, we get *disk B*. The specifications of this disk are given in Table 2.1 and its utilization graph is also drawn in Figure 2.9. This graph shows that in the future the utilization of the bandwidth will be even lower, if the average transfer size remains the same. Note that we are concerned about the utilization of the bandwidth and not the pure transfer rate. Of course, disk B is significantly faster than disk A in terms of speed.

Skeptics may object to our assumption that advancements in the area of mechanical components equal the advancements in data density. Recent history of current disks clearly shows a different trend where data density increased more than the spindle speed and disk-arm speeds. For example, in the last couple of years, the capacity of one platter has gone up from 2 GB up to 20 GB, a tenfold increase. However, the spindle speeds have only doubled from 5400 RPM to 10000 RPM. Advancements in the area of seek times have also not reached this kind of level. Therefore, a disk in which the data density is doubled and the seek and spindle-speeds have only increased 30% may be more realistic.

Surprisingly, however, the utilization graph of such a disk is exactly the same as the graph of disk B. The reason is that any increase in the mechanical components of the disk (i.e., spindle speed, seeks, cylinder and head switches) is lost in the equation. Conceptually, this can be explained as follows. The utilization is determined by the sustained transfer rate and the overhead caused by positioning delays. The sustained transfer rate increases due to faster spindle speeds and lower cylinder and head switch times. Therefore, if the positioning delays remain the same, the utilization of the bandwidth would drop. However, the positioning delays also improve due to faster seeks and lower rotational latencies, which is a result of the faster spindle speed. The net result is that the utilization is unchanged. In short, improvements in the mechanical components have no effect on the utilization, if *all* mechanical components improve *equally*.

If the spindle speed improves at a different rate than the seek and the cylinder and head switches, then the utilization changes, but only a little. An example of this is *disk C*, in which the density has doubled, the spindle speed has increased 50% and the seek and the cylinder and head switch times have decreased only 25%. The specifications of this disk and the corresponding utilization graph are shown in the same Table 2.1 and Figure 2.9. The graph differs only marginally from the graph of disk B. The conclusion that must be drawn from comparing disk B or disk C with disk A is the same: trying to increase the average transfer size will in the future become even more important than it already is today. In other words, we expect that the importance of restricting disk I/O to large sequential transfers will increase in the years ahead.

2.4.2 Disk Bandwidth Utilization

As has been illustrated in Figure 2.9, the straightforward way to improve the effective disk bandwidth utilization is to read and write in large chunks. However, simply increasing the blocksize as the smallest unit of data that can be transferred leads to problems, such as internal fragmentation. Instead, we should try to increase the percentage of disk requests that transfer large amounts of data compared to requests transferring only small amounts.

We do not want to restrict the client to only read and write in large amounts of data. The client should be allowed to transfer the smallest unit of data (one block or even one disk sector). Therefore, it is the task of the disk management layer to somehow convert those small requests into larger sequential disk transfers to improve disk performance.

In the following section we explain how we can increase the number of large data transfers *to* the disk by using a technique called *collective writes*. Next we explain how we can accomplish the same for data transfers *from* the disk by improving the clustering of data using *data reorganizations*.

Collective Writes

One of our objectives is to obtain good write performance by trying to write data to disk in large contiguous chunks. We call such large contiguous chunks **segments**. However, write requests from clients often only write a small amount of data. **Collective writes** is the name for a technique whereby a number of small (random) write requests is turned into one large sequential data transfer to the disk. This transformation is achieved by

accumulating many small writes in main memory (RAM or NVRAM) and by writing a segment full of accumulated data blocks to disk using one large, consecutive, physical write operation.

The fixed size of a segment can be chosen such that a certain desired bandwidth utilization will be achieved when transferring segments. Thus, the exact segment size chosen depends on the characteristics of the underlying disk and on the minimum bandwidth utilization desired for accessing segments at random locations on disk. For example, with a segment size of 256 KB, segment-size data transfers will achieve an average bandwidth utilization of 53% for disk A, and 36% for both disk B and C, compared to 0.4% and 0.2%, respectively, for 1 KB writes.

Collective writes will put the accumulated blocks at another place on disk than their previous versions. This property is nice in at least one respect, as we do not want to have in-place updates (see Section 2.3.1). However, a side-effect of collective writes is that physical clustering deteriorates, which is unfortunate since physical clustering of data is important for read performance (see Sections 2.2 and 2.4.2). In order to restore physical clustering, the data blocks on disk should at times be reordered on disk, which is a process called **data reorganization**. This reorganization is also related to the task of creating consecutive amounts of free space on disk to write segments to. The reorganizations should try to conform to the clustering wishes of the clients, which will be discussed in the next subsection. In short, collective writes form an efficient solution for avoiding in-place updates, but also lead to an increased need for data reorganizations to (re-)enable good performance on sequential reads (see also Chapter 8).

As an example of collective writes and subsequent reorganizations, consider a small update of a large file that is physically clustered on disk. The update only affects a small number of blocks of the file, and these blocks are presented to the disk management layer for writing. Note that the best locations on disk for these blocks will probably be the locations of their previous versions on disk, assuming that those locations had been carefully chosen to provide good clustering. That way, the blocks of the new file will remain clustered.

However, in-place updates are forbidden. Fortunately, accumulating many such small and possibly unrelated updates to different files and subsequently writing them collectively in one or more segments is therefore a good solution. This is because it results in optimal performance when collectively writing these blocks to disk. Note that in this way, the data blocks are protected against crashes quickly and efficiently, but not very well clustered anymore. In order to correct the physical clustering, a number of reorganizations are necessary afterward. How and when the clustering of the data is corrected depends on several factors, and will become more clear in the next section and later chapters.

In some cases reorganizations of the data blocks that are written with a collective write are unnecessary. For example, suppose that a large number of consecutive blocks of a file that must be stored physically clustered, must be written to disk. These data blocks can be written with collective writes in one or more segments to disk.

Segmentwise writes (i.e., writing data to disk via large contiguous disk writes, where each write is the size of a segment) are a reasonably good solution for writing these large amounts of data blocks. In this case there is no strong need for subsequent reorganization, because the data are clustered within segments. In principle, the written segments do not

have to be consecutive on disk. Even if the blocks thus written into these segments form just part of an even larger sequential file, the read performance of this larger file will still remain acceptable. The data of this file are at least clustered in segment-size chunks, which leads to a minimal guaranteed bandwidth.

Of course, it is best to limit the overhead required for reorganizations as much as possible. Therefore, we intend to adapt the way data is actually written to disk depending on the clustering requirements and the type and amount of data blocks that are to be written. In Chapter 5 we present a categorization of data blocks, and discuss how blocks of each category can best be written to disk.

In the discussion on collective writes we have left out one important detail: How do we find or create an empty segment on disk where the collective write can write the accumulated data to? The answer is that the above-mentioned data reorganizer is also responsible for that. Reorganization is the subject of the next section.

Reorganizing

Well-clustered placement of data on disk is very important for read performance, as in that case sequential read access requires less disk-arm movement. Ideally, the successive blocks of each sequentially accessed file should have consecutive locations on disk.

However, this is not an easy task to accomplish, as the creation, deletion, growing, and shrinking of files over longer periods of time lead to so-called **aging**. Aging is the effect where the initial clustering of files deteriorates, and also causes fragmentation, which makes it more cumbersome to allocate a sufficient number of consecutive locations. If file blocks keep their initially allocated physical locations until they are deleted, fragmentation may make the latter even impossible. Note that fragmentation thus may also deteriorate the performance of in particular large writes.

The above illustrates our three main requirements for maintaining good clustering and thus good read performance.

- First, there should be a mechanism to indicate which blocks are related and to what degree they should be physically clustered and/or how urgent it is that the clustering be established. (As a poor man's solution, one could assume that all files require sequential clustering.)

- Second, file blocks should not get a static location on disk, but a dynamic one in order to be able to adapt the locations as required, in particular to counter aging or to meet changing client requirements concerning clustering.

- Third, there should be reorganization software to perform the required adaptations automatically during normal operation. It is especially important that the reorganizations themselves do not make the system unavailable.

We advocate having many small, incremental reorganizations instead of occasionally having one large reorganization. It is often better to regularly do a relatively small reorganization to keep the desired clustering, than to do a relatively large reorganization only once every so often. A really large reorganization is likely to be so work intensive that it may take a relatively long time to complete. In the worst case, the reorganization may

prevent normal read and write requests from completing in parallel and have to wait until the reorganization has finished. Our approach of small, incremental reorganizations in order to minimize the disruptiveness of reorganizations is similar to the use of *incremental garbage collections* as used by some systems to reclaim allocated, but unreferenced parts of memory. Therefore, we advocate an approach whereby small reorganizations can occur during normal operation, and whereby reorganizations are as much as possible performed during idle times, that is, when there are no other disk requests to be serviced.

One real problem left is finding a really good and efficient data storage and reorganization strategy, which is not a trivial task. Here we suggest that one might simply adopt an existing data storage approach, such as the cylinder groups used in FFS. We expect this to be viable, because the segments of the segmentwise storage can be seen as the logical counterparts of the cylinder groups. Having dynamic locations for disk blocks then offers the advantage that reorganizations can help achieve better performance than is possible with more traditional file systems, such as FFS. In current FFS-like file systems aging is not countered automatically. Even small seeks and rotational delays within one cylinder group can already lower performance considerably. However, this is not the main topic of this dissertation. Future work will concentrate on this important aspect.

As reorganizations are required to combat aging anyway, the reorganizations required as a result of collective writes seem not to be an extra problem. However, to get optimal performance one has to make a good choice out of several ways to restore clustering after a collective write. For example, if only part of a file has been written, one can either move the newly written blocks to their original, clustered location or move the nonupdated blocks to the newly written ones or even move all blocks to a third location (see also Chapter 8).

2.4.3 Synchronous Writes

The second performance problem we identified is the use of synchronous writes. In general, synchronous writes have a disastrous effect on performance, so they should be avoided as much as possible. Nevertheless, many file systems still use synchronous writes in an attempt to prevent file system corruption due to crashes.

A file system contains both client data as well as metadata. To a file system metadata is much more important than client data since it defines the structure of the file system. If metadata is corrupted, the file system is inconsistent, which could cause client data to become inaccessible or even inadvertently overwritten. In either case, client data is lost.

In order to preserve metadata consistency, file systems should be careful when updating metadata. Especially when multiple updates to metadata are necessary to bring the file system from one consistent state to another. A typical example of such a case is the creation of files in a UNIX file system, where both an i-node and a directory block must be updated. The i-node must be updated to indicate that it is in use, and the directory block should be updated to hold a reference to that particular i-node because a new file is created in that directory. If a crash occurs after writing the first update to disk, but before the second update could be written, the file system would be left in an inconsistent state.

The way some file systems solve this problem is by minimizing the impact this inconsistency could have on the file system. For example, if in our example only the i-node

update reached the disk, then the i-node would exist, but no directory would point to it. For the file system this situation would mean that the i-node is in use, but is inaccessible. This inconsistency is acceptable because, other than the loss of an i-node, the file system still behaves normally. However, what if the directory was updated, but the i-node did not reach the disk? Then the directory would hold a reference to an i-node that is not in use and thus not correctly initialized. This inconsistency is clearly unwanted. The situation could even become worse when that i-node is subsequently allocated during the creation of another file. This inconsistency is certainly worse than the previous inconsistency. Therefore, in the case of file creation, UNIX file systems usually write the i-node to disk before the directory block. For all multiple metadata updates such an order is specified, so that crashes would affect the normal behavior of the file system as little as possible. Any inconsistencies left in the file system after a crash can be detected and corrected by a scavenger program such as `fsck` [Kowalski, 1978]. For example, this program could correct the situation of the inaccessible i-node as described in the beginning of this paragraph by freeing the i-node.

However, due to disk scheduling policies and write caching, ensuring the correct write order is not as simple as issuing the writes in the desired order to the disk. The way that UNIX file systems often enforce this order is by using **synchronous writes**. A write request is sent to the disk only after the previous write request has completed and has safely reached the disk. Because of the high price to be paid in performance caused by waiting for the request to complete, synchronous writes are normally used only to safeguard crucial metadata on disk in some precise order. Client data are written to disk asynchronously, since loss of client data is not crucial for file system consistency.

A system failure can still result in an inconsistent file system. However, the assumption of this scheme is that due to the synchronous writes, the damage caused by the inconsistent state is small. Furthermore, that inconsistent state is easily detectable and correctable. Unfortunately, these assumptions are often untrue. Severe file inconsistencies and data losses are still possible, in particular because of the presence of in-place updates in most file systems. A crash during a synchronous write that uses an in-place update to write a metadata block can still destroy the contents of that metadata block on disk and cause enormous wreckage in a file system. In summary, one could say that the loss in performance for synchronous writes is very high, and yet the protection offered is far from complete.

In literature, many solutions have been proposed to avoid using synchronous writes to maintain data integrity. Examples are NVRAM, a metadata log, or soft-updates [McKusick and Ganger, 1999]. As we described in Section 2.3, we prefer a disk management layer to support atomic multiblock writes (in the form of ARUs) or even block-level transactions. If used correctly, this does not only make synchronous writes superfluous, but also enables full data integrity for all data; that is, metadata as well as client data.

2.5 Summary

In this chapter we have looked at some problems relating to disk usage in more depth. After the discussion of each problem we have outlined some possible solutions. When

properly implemented these solutions will result in a disk management layer that can provide its clients with improved modularity, data integrity guarantees and improved performance. The solutions discussed above can be seen as minimum requirements for our Logical Disk. For ease of reference, we summarize them here again with references to where they have been discussed. Note that these requirements are not unrelated; some requirements are logical consequences of others.

(1) Modularity (Section 2.2)

(2) Location transparency (Section 2.2)

(3) Mechanism to express requested data block clustering (Section 2.2)

(4) Mechanism to express read-ahead (Section 2.2)

(5) Avoidance of in-place updates (Section 2.3.1)

(6) Mechanism to indicate essential command-ordering requirements; avoidance of visible command reordering when a specific ordering is requested (Section 2.3.2)

(7) Atomic Recovery Units (2.3.3)
(or possibly even block-level transactions (Section 2.3.4))

(8) Collective writes (Section 2.4.2)

(9) Automatic data reorganization in background (Section 2.4.2)

In the next chapters, we will introduce the Logical Disk and discuss how it meets these requirements. Requirements 1, 2, 3, 4, 6, and 7 are dealt with when we present the external interface of LD in Chapter 3, the others are more implementation related and are discussed in Chapters 4 – 8.

Chapter 3

Programming Interface

In the previous chapter we have analyzed some problems of current disk usage, and identified solutions to these problems. The chapter concluded with summarizing a number of requirements that need to be satisfied in a complete solution. In this and the following chapters we discuss the **Logical Disk (LD)**, which is our integrated solution that fulfills these requirements. This chapter focuses on the external part of LD, such as the supported abstractions. The following chapters discuss LD's internal design and explains some implementation issues in detail.

We frequently use the term *clients of LD* or simply *clients* to refer to software that runs directly on top of LD, such as a file system or DBMS. We use the term *client data* to refer to data stored on disk by such clients of LD, including both their data and metadata. Likewise, we use *metadata* to refer to the data that LD keeps to maintain its own administration (see also Section 4.1).

3.1 Goals of the Logical Disk

As stated in the previous chapter, our approach to solving the identified data integrity problems and alleviating the identified performance problems is the creation of a new software layer that provides a new disk abstraction. This software layer is located between the disk driver software and the DBMS and/or file system (FS) software. We refer to this new layer as a *disk management system*.

With the introduction of the Logical Disk, we want to create a separate layer of software that improves the use of disk hardware. In particular, the Logical Disk tries to improve the quality of the disk system software and the quality of the storage that such a system offers, and also tries to improve the performance of a disk system. LD improves the quality of the software by tackling the lack of modularity in current storage system designs (Requirement 1, at the end of Chapter 2). The quality of the storage functionality of disk systems is improved by solving the data integrity problems of such systems. These quality improvements refer to our desire to develop better disk management software. In quantitive respect LD tries to improve the performance of current disk systems by focusing especially on the disk bandwidth usage.

Unfortunately, it is likely that improved data integrity will come at a cost. In this research we have restricted ourselves to software solutions, and as a result the cost may well be a loss in performance. Adding another software layer to an existing architecture will undoubtedly have its price in performance. As is often the case, there is a kind of trade-off between improving data integrity and improving performance. Although the word trade-off suggests that both factors are not attainable at the same time, we also think that it is possible to find a compromise that improves data integrity without significantly degrading performance. However, we have decided that in the process of finding this compromise our first priority concerns solving the data integrity problems because we feel that the integrity of data is crucial to a storage system.

In short, with LD we try to:

(1) Improve the modularity of storage management software,

(2) Improve the functionality of disk storage subsystems by improving in particular their data integrity properties, and still

(3) Provide performance competitive to other systems.

3.2 Storage Abstractions

The main functionality of a disk management system, such as LD, is to allow clients to store data in a persistent way on a hard disk. LD does not grant the client direct access to the hard disk, but provides a number of storage abstractions that can be used to store data on disk in a structured manner. These abstractions act as data containers and are discussed in this section.

3.2.1 Logical Blocks

The unit of storage that LD supports is the **logical block**. The size of a logical block is currently set to the smallest possible unit that a disk can read or write: a sector (usually 512 bytes). These logical blocks hold the data that clients want to store on disk. To LD the data in a logical block is an unstructured array of exactly 512 bytes.

LD can transfer data only in sizes which are multiples of the logical block size. It is the task of the client of LD, such as a file system or a DBMS, to provide support for data transfers of an arbitrary size. This situation is similar to how current storage systems work: the disk and disk driver support only blockwise I/O, while the file system or DBMS allows an arbitrary size of data to be stored and retrieved.

3.2.2 Disk Files

As a logical block is only one sector, clients have fine-grained control over the amount of physical disk space they use. This allows a client to minimize internal fragmentation. However, most data objects need more space than fits within one logical block. Such a data object can be stored by distributing its data over multiple logical blocks. To relieve

the client of the burden of administrating these blocks, LD offers the concept of a *disk file* that enables the client to indicate a relationship between a group of logical blocks.

A **disk file** is a sequence of zero or more logical blocks that can be addressed as a single object. More precisely, a disk file is a container that can hold logical blocks in a one-dimensional array. Each block in a disk file is individually accessible and manipulatable. The position of a logical block within a disk file is indicated by an **offset** (or index) in the array. The disk file can be sparsely populated with nonempty blocks similar to a UNIX file-system file, which can contain holes.

A typical use of a disk file is to store the data of one object. As an example of the use of a disk file consider a file system that runs on top of LD. Such a file system can be designed to use one disk file to store the data of a single file-system file. In current file systems, the administration of the disk addresses of the physical blocks that belong to a file-system file are kept by the file system in special structures. For example, a UNIX file system uses *i-nodes* and a DOS file system uses the *File Allocation Table (FAT)* for this purpose. However, in a file system on top of LD, most of this administration is already maintained by LD. Access to a particular block of a file is possible by specifying the disk file and the offset of the block within that disk file to LD. Therefore, as LD already takes care of the disk block administration for the disk files, it is not necessary for the file system to also keep such a block administration, which reduces the complexity of a file system.

3.2.3 Disk File Headers

Most file systems keep more meta-information about each data object (e.g., file or directory) than just its block administration. Typical examples of such meta-information include the owner of the data object, access control information, size, etc. In a UNIX file system this meta-information is stored in the *i-node* of a file. A file system on top of LD can store the data object in a disk file, but where can it store this kind of meta-information? There are at least three alternatives to store such meta-information. The first alternative is to store the meta-information together with the actual contents of the object that are stored in the disk file. For instance, the file-system meta-information can be stored within the first few blocks of the disk file.

The second alternative is to use a separate disk file to hold the file-system meta-information. For example, a file system could store each i-node in a separate disk file or store multiple i-nodes together in one disk file, which is more efficient. Consider, for instance, a file system that stores a file-system file in a disk file. The directory information of each directory, which is the list of file names and corresponding i-node numbers of files in that directory, is stored in a separate disk file. The i-nodes of the file-system files in that directory are stored together with the directory information in the same disk file. This way, the i-nodes of files in the same directory are all stored clustered in one disk file, and are immediately accessible with the directory information. This technique is known as **embedded i-nodes** [Ganger and Kaashoek, 1997].

However, the data of the object itself and its meta-information have different characteristics. An object's meta-information is often small and accessed relatively often. For example, in a file system common actions are browsing through the file system with the help of a file browser or listing directory entries. Both actions access the meta-information of

files in directories without accessing the actual contents of these files. Therefore, we have chosen to let LD also provide direct support for another way of storing meta-information, which creates a third alternative of storing meta-information.

In LD each disk file has a **disk file header**, or simply **file header**, that can contain a small amount of data. A typical use of this file header is to store file-system meta-information about the data stored in the associated disk file. Each file header has a maximum size in the order of one logical disk block. This size is more than enough to store the typical i-node information in the file header. LD stores file headers together with LD's own metadata. This metadata contains, for example, the disk block administration that is kept for each disk file. This way, accessing a file header of a disk file is more efficient than accessing data in logical data blocks of the disk file. The file header is not intended as an alternative storage location for the disk file, which can store much larger objects. However, in some cases it may be efficient to store client data in the file header and thus implement **immediate files** [Mullender and Tanenbaum, 1984].

In summary, LD provides the following three ways for clients to store their meta-information. First, the meta-information can be stored together with the data it describes by storing it in the same disk file as the data. Second, the meta-information can be grouped together by storing them separately in one or more disk files. Last, the client can choose to let LD handle the file-system's meta-information as part of LD's own disk block administration, which is LD's metadata, by using file headers.

3.2.4 Disk Clusters

The idea behind a disk cluster is somewhat similar to the idea behind a disk file. A disk cluster is used to indicate a logical relationship between several disk files. A **disk cluster** is a sequence of zero or more disk files that can be addressed as a single unit. Similar to a disk file, the disk cluster is a container that can hold disk files in a one-dimensional array. It is not possible to nest disk clusters. Disk files and disk clusters can help the client to structure its data on disk. For example, the same file system in our previous example could store all the files within one directory in one disk cluster. Analogous to disk files, each disk cluster also has a separate storage space for meta-information associated with it: a **disk cluster header**, or simply **cluster header**. A file and a cluster header are quite similar, and we will simply use the term *header* to refer to both kinds.

3.2.5 Managing Blocks, Disk Files and Disk Clusters

The client can create and delete logical blocks, disk files and disk clusters by calling the appropriate functions of LD. However, there are some restrictions. A logical block cannot exist on its own, it must be contained within exactly one disk file. Analogously, a disk file cannot exist on its own, it must be contained within exactly one disk cluster.

Disk clusters and disk files must be explicitly created and deleted by a client. First, the client has to create a disk cluster by calling the function `ld_create_cluster`. After the creation, the client has an empty disk cluster, that is, a cluster that contains no disk files. Subsequently, the client can create a disk file within that cluster by calling `ld_create_diskfile`, which creates an empty disk file within a given cluster. The exact

prototypes of these and other functions of LD, including their parameters, are given in following sections.

Logical blocks within the disk file are implicitly created when the client writes data to a certain block position in a disk file using the function `ld_write_data`. The client specifies the starting position of the logical block(s) within the disk file where the data should be stored and supplies the data. LD implicitly creates the blocks at the requested positions and fills them with the supplied data. If a logical block already existed on a requested position, its contents are simply replaced with the new data. In this last case, LD will make sure that the old contents are not overwritten in-place, which would violate our no in-place update requirement. We will deal with this issue in the next chapter.

Logical blocks must be deleted explicitly by calling `ld_delete_blocks`. This function deletes a range of blocks within one disk file, which frees the associated disk space. A disk file and a disk cluster can be deleted by calling `ld_delete_cluster` and `ld_delete_diskfile`, respectively. Deleting a disk file automatically deletes all the blocks within that disk file and the corresponding file header; deleting a disk cluster deletes all disk files within that disk cluster and the corresponding cluster header.

The creation of a disk cluster or disk file also automatically creates the corresponding cluster and file headers, which are empty at the start. These headers can be manipulated via the `ld_set_fh`, `ld_get_fh`, `ld_set_ch`, and `ld_get_ch` functions. In contrast to the size of a logical block, the size of a file or cluster header is variable. When writing a header, the client must, therefore, explicitly specify how large the header is. Any previous contents of the header are replaced by the new contents. Writing a header of size 0 has the effect of deleting an existing header. `ld_get_fh` and `ld_get_ch` reads the specified number of bytes from the start of a header. LD will not return more bytes than the size of the actual header, even if more bytes are requested in the parameter of the function call. Specifying the maximum size that LD allows for a header will, therefore, always correctly return the complete header. The actual size of the returned header is put in the return value of the function.

3.2.6 Addressing

Every disk cluster has a **cluster identifier**, which is a positive integer, unique within LD. The client chooses the cluster identifier, cluster_id for short, when creating the cluster by passing it as an argument to the function `ld_create_cluster`. If the supplied identifier is already in use and, therefore, not available, LD will return an appropriate error. The advantage of this scheme, where the client supplies the cluster_ids instead of having LD assign them, is that the client has control over the use of the address space of available cluster_ids. This scheme allows the client to structure the way it uses cluster_ids and thus add semantics to the cluster_ids. Note that if multiple clients use LD concurrently, they should agree on the semantics chosen for cluster_ids or partition the address space such that each client has access to its own range of addresses, so that the clients do not interfere with each other.

The function prototypes of `ld_create_cluster` and other functions are listed in Listing 3.1. There are two more arguments that need to be supplied to these functions: `stream` and `aru`. These arguments will be explained in Sections 3.4 and 3.5 where we explain

```
/* Create and delete a disk cluster with a given cluster_id . */
int  ld_create_cluster (uint32_t stream, uint32_t aru, uint32_t cluster );
int  ld_delete_cluster (uint32_t stream, uint32_t aru, uint32_t cluster );

/* Create and delete a disk file with a given cluster_id and diskfile_id . */
int  ld_create_file (uint32_t stream, uint32_t aru, uint32_t cluster, uint32_t file );
int  ld_delete_file (uint32_t stream, uint32_t aru, uint32_t cluster, uint32_t file );

/* Read, write and delete a given number of blocks starting at a certain
 * position in a given disk file . The buffer will hold the data that
 * are read from disk or holds the data that are to be written to disk . */
int  ld_read_data (uint32_t stream, uint32_t aru, uint32_t cluster, uint32_t file,
                   uint32_t offset, uint32_t count, char *buf);
int  ld_write_data (uint32_t stream, uint32_t aru, uint32_t cluster, uint32_t file,
                    uint32_t offset, uint32_t count, char *buf);
int  ld_delete_blocks (uint32_t stream, uint32_t aru, uint32_t cluster,
                    uint32_t file, uint32_t offset, uint32_t count);

/* Set or read the contents of a file or cluster header. On success, the read will
 * return the actual number of bytes read from the file or cluster header. */
int  ld_set_fh (uint32_t stream, uint32_t aru, uint32_t cluster, uint32_t file,
                uint32_t writesize, char *data);
int  ld_get_fh (uint32_t stream, uint32_t aru, uint32_t cluster, uint32_t file,
                uint32_t readsize, char *data);

int  ld_set_ch (uint32_t stream, uint32_t aru, uint32_t cluster
                uint32_t writesize, char *data);
int  ld_get_ch (uint32_t stream, uint32_t aru, uint32_t cluster
                uint32_t readsize, char *data);
```

Listing 3.1: Mapping Interface: read, write, delete headers and addresses.

command streams and *atomic recovery units*, respectively. All functions return some error code on failure.

Each disk file has a **disk file identifier**, diskfile_id for short, which is also a positive integer, that is unique within its cluster. The combination of **cluster_id** and **diskfile_id** uniquely identifies a disk file within LD. Creation of a disk file is similar to the creation of a disk cluster. The client supplies a diskfile_id and the cluster_id of the cluster in which the disk file is to be created. The cluster must already exist and the diskfile_id within that cluster must still be available. If either condition is not met, LD will return the appropriate error.

Again, the client has control over the usage of diskfile_ids, which allows the client to add semantics to them. For instance, consider a file system on top of LD that stores its files of a single directory in disk files within one disk cluster. This file system can reserve a special diskfile_id (e.g., 1) within each disk cluster to hold the directory mapping and reserve another (e.g., 2) to hold the i-nodes of the files.

A block within a disk file is uniquely identified by its **offset** within the disk file. The

```
/* Find the previous or next unused cluster_id , diskfile_id or block offset .
 * The search backward or forward is started from a given start point .
 * If found , the answer is stored in the parameters nclusterp , nfilep and noffsetp .
 * Otherwise , an error is returned . */
int ld_prev_unused_cluster (uint32_t stream , uint32_t aru ,
                            uint32_t cluster , uint32_t * nclusterp );
int ld_next_unused_cluster (uint32_t stream , uint32_t aru ,
                            uint32_t cluster , uint32_t * nclusterp );

int ld_prev_unused_file (uint32_t stream , uint32_t aru ,
                         uint32_t cluster , uint32_t file ,
                         uint32_t * nclusterp , uint32_t * nfilep );
int ld_next_unused_file (uint32_t stream , uint32_t aru ,
                         uint32_t cluster , uint32_t file ,
                         uint32_t * nclusterp , uint32_t * nfilep );

int ld_prev_unused_block (uint32_t stream , uint32_t aru ,
                          uint32_t cluster , uint32_t file , uint32_t offset ,
                          uint32_t * nclusterp , uint32_t * nfilep , uint32_t * noffsetp );
int ld_next_unused_block (uint32_t stream , uint32_t aru ,
                          uint32_t cluster , uint32_t file , uint32_t offset ,
                          uint32_t * nclusterp , uint32_t * nfilep , uint32_t * noffsetp );

/* Find the previous or next used cluster_id , diskfile_id or block offset .
 * The search backward or forward is started from a given start point .
 * If found , the answer is stored in the parameters nclusterp , nfilep and noffsetp .
 * Otherwise , an error is returned . */
int ld_prev_used_cluster (...);
int ld_next_used_cluster (...);

int ld_prev_used_file (...);
int ld_next_used_file (...);

int ld_prev_used_block (...);
int ld_next_used_block (...);
```

Listing 3.2: Find used and unused cluster_ids, diskfile_ids or offsets.

offset is also a positive integer. Every logical block has a unique **logical block address**, which, in our current design of LD, is a 12-byte triplet consisting of three 4-byte unsigned integers: cluster_id, diskfile_id, and offset within the disk file. We have chosen to use 4-byte unsigned integers, so that there is little chance of running out of possible identifiers. With 4-byte identifiers LD can support over 4 billion clusters, each with over 4 billion disk files. With a 4-byte offset and logical blocks of 512 bytes, the maximum size of each disk file is 2 TB. In the future, this may be too small and we may need to upgrade to 8-byte unsigned integers, which poses no significant problem as the choice for 4-byte unsigned integers is not fundamental to LD's design.

Clients can access their data on disk only using these logical addresses. This fulfills the *location transparency* requirement (Requirement 2, on page 42). The logical addresses are internally mapped onto physical addresses. As a consequence, applications are freed from the need to administrate and maintain physical disk block addresses. Furthermore, location transparency enables LD to dynamically adapt the clustering of files on disk. In LD, data blocks do not have fixed locations on disk, unlike the situation in most traditional file systems. The mapping of logical addresses to physical addresses is maintained by LD in a table called the **Mapping**, which will be discussed in the next chapter.

To facilitate the search for available (i.e., unused) cluster_ids, LD provides the function `ld_next_unused_cluster`, which scans the range of cluster_ids in increasing order starting at a given start cluster_id and returns the first *unused* cluster_id it encounters. Similarly, the function `ld_prev_unused_cluster` finds unused cluster_ids in the other direction (smaller cluster_ids). There are also similar functions to seek unused diskfile_ids within a cluster and to seek unused block positions (offsets) within a diskfile. In addition to these functions, there are also functions to seek *used* cluster_ids, diskfile_ids and block positions. All these functions are listed in Listing 3.2. For brevity, the parameter lists for some functions are left out in the listing, but these are similar to the ones that are shown in the listing.

3.3 Physical Clustering

Another requirement we have identified is the ability to express physical clustering wishes to LD (Requirement 3, on page 42). *Logical* relationships between blocks can already be expressed in LD by disk files and disk clusters. Additionally, we also need a way to indicate that blocks need to be stored close together on disk for performance reasons, which is a *physical* relationship.

We have chosen to reuse the disk file and disk cluster as the mechanism for physical clustering. On request, LD will physically cluster the blocks of a disk file, which we refer to as **intrafile clustering**. In this case, LD tries to physically store the blocks in order of their offset since this optimizes sequential access to the disk file. The client must decide for each individual disk file whether physical clustering is important. Similarly, the client can request that the disk files in a single disk cluster are stored physically close together on disk, which we refer to as **interfile clustering**. The functions to request physical clustering for disk files and disk clusters in LD are listed in Listing 3.3.

Naturally, physical clustering requires some effort from LD to keep blocks clustered on disk over time. Therefore, the overall performance of the whole system benefits if physical clustering is requested only for disk files and clusters that actually need it. Recall that a client of LD is not a human user, but a software layer, such as a file system, potentially serving many human users. Whereas human users are prone to bias, and therefore, likely to request clustering for all their own data in the expectation that this will increase their read performance, it is the task of a file system to optimize the *overall* performance of the system. Therefore, the file system should be the final authority that determines for each individual disk file or cluster whether to actually request physical clustering. A file system could base its decisions on information acquired by monitoring access patterns.

```
/* Set or unset clustering for a disk file or disk cluster. */
int    ld_set_clustering_cluster ( uint32_t stream, uint32_t aru,
                                   uint32_t cluster, boolean_t clustering_request );

int    ld_set_clustering_file ( uint32_t stream, uint32_t aru,
                                uint32_t cluster, uint32_t file,
                                boolean_t clustering_request );
```

Listing 3.3: Set or unset clustering for a disk file or disk cluster.

These access patters will show which disk files and clusters are often accessed sequentially, and therefore, would benefit from physical clustering. Additionally, to help a file system make its decisions, human users could provide *clustering hints* to the file system.

The actual process of clustering is transparent to the client. LD only gives a best-effort guarantee when trying to cluster blocks on disk. Maintaining the clustering on disk may involve a significant amount of I/O operations to move blocks on disk. LD must find a reasonable balance between fulfilling the clustering wishes of clients and responding quickly to normal read and write requests of clients. Both tasks require disk bandwidth, which is a limited resource. We will discuss clustering and reorganizations further in Chapters 4 and 8.

By choosing the disk file and disk cluster as the physical clustering mechanism, we do put some limitations on clustering. For example, it is not possible to cluster individual blocks from different disk files. We do not expect that this limitation is too restrictive since we expect that disk files will be used in such a way that the blocks in a disk file store the data of a single object. It is not uncommon that such an object is often accessed in a sequential manner, so that physical clustering is desirable. For this reason, LD provides support for physical clustering of the blocks of a disk file. However, it is in general less likely that individual blocks from different disk files are accessed sequentially; therefore, the lack of support to cluster such blocks is not greatly missed. Still, if such clustering is somehow desirable, the client can create these disk files in the same cluster, and ask LD to cluster the disk files of that cluster. That way, the disk files of those blocks will be stored close together on disk.

Another limitation is that interfile clustering is only possible for disk files in one disk cluster. For example, suppose a file system on top of LD uses a disk file to hold a file system file and uses a disk cluster to cluster the files in one directory. Now consider what would happen if a user of this file system moves a file from one directory to another. The desired behavior would be for LD to cluster the blocks of that file with blocks of files in the other directory. However, the logical address of the disk file remains the same, so LD will still cluster that file with files of the old directory. In order to get the file to be clustered with files in the new directory, its logical address should change. Changing logical address of the disk file is of course undesirable since there may be many references to this disk file in the file system, which should then all be updated.

The previous examples make it clear that the logical addresses pose limitations on the way blocks can be clustered. These limitations arise because a logical address contains information about the clustering information of the corresponding disk block(s), if phys-

ical clustering has been requested. This case shows that logical addresses are not ideal object identifiers [Wieringa and de Jonge, 1995]. This is a consequence of our choice to reuse the disk file and disk cluster as our physical clustering mechanism.

Ideally, LD should allow clients to specify *logical* and *physical* relationships between blocks, disk files or disk clusters independently of each other. The reason is that the principle of separation of concerns is once more applicable. Unfortunately, treating logical and physical relationships as independent properties leads to a double administration, which leads to more complexity within LD. Therefore, we have decided not to include such a scheme in our current design, as yet.

In a future release of LD, we plan to implement a more general clustering mechanism, which allows any two disk files to be clustered. This more general scheme still does not allow any two individual blocks to be clustered, but we feel that the practical usefulness of this last feature is questionable. Additionally, the implementation of independent clustering on a per block basis potentially requires an extensive amount of administration.

3.4 Command Streams

Logically, clients send successive read and write commands to LD via a channel, called a **command stream** or just a **stream**. LD guarantees that the commands issued by a client in a command stream will be executed in such a way that it seems as if they have been executed in the order of their arrival. LD can perform optimizations to improve performance, as long as they are consistent with preserving this ordering guarantee. Streams in LD fulfill Requirement 6, on page 42, which states that a client must be able to control the execution order of commands, when necessary. In particular, a client may not be confronted with more complex recovery because of invisible command reordering from disk scheduling algorithms.

A client can open multiple command streams to LD in order to achieve pseudo-concurrency, and thus reach a higher degree of concurrency. Each command stream has a separate **command stream identifier** or stream_id for short, which is currently a 1-byte integer. LD only makes guarantees about the execution order of commands within the *same* stream. LD makes no guarantees about the execution order of two concurrent commands in *different* command streams. This mechanism gives the client the flexibility to control the execution order of commands whenever that is important to maintain data integrity, but it also allows the client to give LD the freedom to schedule commands in order to improve performance.

In the previous chapter we opted for a solution that allows the client to specify ordering requirements for its commands to the disk management layer. LD uses streams to fulfill this requirement. The client can send a group of commands on the same stream if it wants to control their execution order, and it can sent them on different streams if the commands are to be executed independently of each other.

In principle, it is possible that multiple clients share a single stream. However, since a client of LD is typically a file system or database system, it is unlikely that two such clients would want to share a stream. Each client typically has its own collection of data objects that it manages. Commands sent by a client will likely only access that client's

data objects and are, therefore, independent of the commands from another client. The most logical option is thus for different clients to sent commands on different streams, to optimize the concurrency degree.

In contrast to the situation of multiple clients sharing one stream, it is likely that one client uses multiple streams. For example, a file system may want to execute independent commands which access different data objects concurrently to improve performance. This can be accomplished by sending these commands on different streams. It may be tempting to create one stream for each file-system user that is active in a file system. However, when the commands are not independent, but access the same data objects, then concurrency control is needed to guarantee correct execution and to maintain data integrity.

Unfortunately, LD does not support full transaction semantics yet. In other words, in the current version of LD concurrency control needs to be done by the clients. A group of commands that act on the same set of data blocks and are sent on different command streams to LD must, therefore, be serialized by the client. For example, clients can use a locking mechanism to serialize access to the underlying data objects. In addition to a locking mechanism, other mechanisms are also needed to provide full concurrency control, such as facilities to solve or avoid deadlock situations, which complicates matters. In the current design LD does not yet provide any special support for locking or any other concurrency control method, but this is planned for a future version.

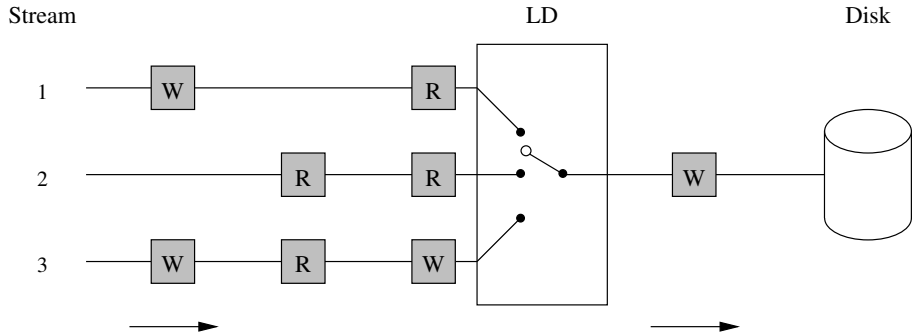


Figure 3.1: Streams in LD.

Note, that to get an actual serializable execution of concurrent commands it is not enough to send a serializable or even a serial schedule to LD when the commands are sent over multiple streams. The execution order of concurrent commands in different streams is undetermined, so the execution order is not guaranteed to be the order that a client has determined in its serializable schedule. The way to guarantee serializable execution is by sending a serializable schedule over a single stream. If multiple streams are used, an external mechanism, such as locking, must enforce the serializable order of execution by synchronizing when to send commands on each streams. We will return to the subject of serializable execution of concurrent commands in the following section on Atomic Recovery Units.

```
/* Create and delete a stream. */
int ld_create_stream (uint32_t *stream);
int ld_delete_stream (uint32_t stream);

/* Flush a stream. After this function has returned, all committed data
 * within this stream are recoverable. */
int ld_flush (uint32_t stream);
```

Listing 3.4: Create and delete streams.

An example of multiple streams is given in Figure 3.1. It shows LD in the middle with three streams, with stream identifiers 1, 2, and 3. The gray boxes on the streams represent commands that have been sent in the streams. The commands are marked with a letter denoting whether the command is a **R**ead or a **W**rite command. LD serves the commands by picking a command from the front of a stream. LD is free to choose which stream it serves next. Of course, a reasonable requirement is that LD schedules the streams in a fair manner to avoid starvation.

A client must explicitly create and delete a stream with the functions listed in Listing 3.4. `ld_create_stream` creates a stream and assigns a stream_id to it. If an error occurs during creation, an appropriate error is returned. After this call succeeds, the client can send commands on this stream by passing this stream_id to calls of LD. `ld_delete_stream` will delete a given stream. The function call to create a stream is sent outside a stream. LD does not guarantee anything with respect to the execution order of this command compared to other commands. The command to delete a stream is sent on the stream that is to be deleted. So, a `ld_delete_stream` is always the last command on a stream. It is also possible to flush a stream. `ld_flush` forces LD to flush the contents of its internal buffers for a particular stream to disk so that they become recoverable. The function prototype of this flush function is also given in Listing 3.4.

A simple use for multiple streams is to associate a separate stream to each separately stored file system[†]. For example, if in a UNIX system the directories root (`/`) and `/usr` are two different mounted file systems, then commands sent to one file system are independent of the commands sent to the other. Therefore, to increase concurrency, these commands can be sent in two different streams.

Another example for using multiple streams is making a backup of a file system while the system is running. The backup-process will read all data from disk and those commands can be sent on separate streams to increase performance. However, to make sure that the backup-process makes a consistent snapshot of the entire system at a certain moment in time, locking or some other form of concurrency control is needed.

[†]It is unfortunate that the word *file system* is commonly used both to refer to the data stored on disk in a certain file-system-specific format (e.g., the result of executing the command `mkfs`) and to the part of the kernel responsible for accessing files and directories on disk.

```
/* Start , commit, and abort an ARU. */
int  ld_start_aru  ( uint32_t  stream , uint32_t *arup );
int  ld_commit_aru( uint32_t  stream , uint32_t  aru );
int  ld_abort_aru  ( uint32_t  stream , uint32_t  aru );
```

Listing 3.5: Start, commit and abort ARUs.

3.5 Atomic Recovery Units

Each and every command sent to LD is executed atomically. In the previous chapter, we also identified a requirement for atomic multiblock writes (Requirement 7, on page 42). LD satisfies this requirement by supporting **Atomic Recovery Units (ARUs)**. An ARU enables the client to group multiple commands that are sent to one stream into one atomic action. ARUs are not the same as transactions. Transactions have the properties of *atomicity*, *isolation*, and *durability*, as explained in Section 2.3.4. In contrast, ARUs only guarantee that after a crash either all or none of the updates within a completed ARU are recovered (i.e., they guarantee *atomicity*). For now, ARUs do not support *isolation*, that is, do not support *concurrency control*. Consequently, the effect of concurrent updates to blocks using different streams is undefined. Furthermore, LD provides only limited *durability*, which means that a completed ARU does not yet guarantee that the updates of that ARU are recoverable. However, a client can follow the commit call by a flush call to tell LD to flush its dirty buffers to disk, thereby making the ARU recoverable.

To create ARUs consisting of multiple commands, the client must explicitly start and end (commit) the ARU by calling the corresponding functions `ld_start_aru` and `ld_commit_aru`, respectively. After starting an ARU, LD assigns an **ARU identifier** or aru_id to the ARU. An aru_id is a positive integer, that is used to uniquely identify a running ARU within LD. This aru_id should be supplied to future commands sent on the same stream to LD to indicate that those commands belong to the same group of commands that must be executed as one atomic action. Currently, LD sets the aru_id equal to the stream_id in which the ARU was created. Since each stream can at most have one running ARU at a time (see below), stream_ids are sufficiently unique to identify all concurrently running ARUs within LD (see below). The function `ld_abort_aru` can be used to abort an already started ARU. The updates of an aborted ARU are discarded. The functions are given in Listing 3.5.

Semantically, a single command sent *outside* an ARU is equivalent to an ARU that consists of only one command. The only difference is that the start and commit calls are implicit. It is sometimes convenient to think of a command sent outside an ARU as a single command ARU, or a **simple ARU**. We will use the specific term **composite ARU** to refer to a multi-command ARU, when necessary. To run a command of LD as a simple ARU, the client can call the corresponding function and supply an aru_id of 0.

There are some restrictions to the use of ARUs. Each ARU is strictly tied to one stream and each stream can have only one active ARU at a time. Therefore, ARUs must be started and committed within the same stream and it is not possible to group commands sent on different streams in one ARU. Nor is it possible to nest ARUs within one stream. Note, that this last restriction also means that once a composite ARU has started no simple

ARUs can be sent on the same stream until that composite ARU has been committed. All commands described in this chapter are allowed within a composite ARU, except for the call `ld_create_stream`. Sending the call `ld_delete_stream` on a stream within a running composite ARU will implicitly abort the ARU before deleting the stream.

The blocks on the disk that are visible to all streams are called **committed** blocks. Conceptually, each ARU sees all the committed blocks of the disk. However, all updates that are made within an ARU are tentative until the ARU commits. Each update creates a private copy of the updated block(s). This copy is private to the ARU and the stream in which the ARU was created, and are, therefore, not visible to other streams. These tentatively updated blocks are called **uncommitted** blocks. In other words, an ARU running in a stream can see all committed changes that were made in either this or any other stream, and can also see all uncommitted changes that were made within that ARU. Note that we use the word 'uncommitted blocks' not only to refer to blocks that are updated or newly created within an ARU, but also to blocks that are deleted within an ARU. Internally, LD uses a *copy-on-write* technique to minimize the amount of copying.

The commit of an ARU turns all its uncommitted blocks atomically into committed blocks. After this commit these updated blocks are immediately visible to other streams and ARUs. As a result every ARU can always access the most recent committed version of any block, even if that block has been committed (by another ARU) after this ARU has started. Only if the ARU has updated a block itself, will it see the private uncommitted version, instead of the committed version, as is to be expected. An abort of a running ARU must undo all its changes, which means discarding its uncommitted blocks.

As an example of the intended use of ARUs, consider the creation of a file in a UNIX-like file system. This operation requires a single atomic update to both an i-node and a directory entry. Consider what happens if the UNIX-like file system is built on top of LD. Suppose that such a file system stores each file-system file in a disk file, and the i-node information of a file is stored in the corresponding file header of the disk file. Furthermore, suppose that all the files in a directory are stored in a single disk cluster. Finally, the directory information, which stores the mapping between file name and diskfile_id, is stored in the first diskfile (diskfile_id 1) of the cluster. In this setup, the creation of a new file system file within an ARU involves the following actions:

(1) `ld_start_aru(...)` : start an ARU

(2) `ld_read_data(...)` : read directory information from the first disk file of the cluster

(3) `ld_create_diskfile(...)` : create a new disk file in the cluster

(4) `ld_write_fh(...)` : write the i-node information of the new disk file

(5) Add a new directory entry for the new file to the directory information, which was read into main memory in step 2

(6) `ld_write_data(...)` : write the updated directory information back to disk

(7) `ld_commit_aru(...)` : commit the ARU

For clarity, we have left out a number of actions, such as the lookup of the directory through the file system tree, checking whether the file-system user has the proper permissions to create this file-system file, or finding a suitable diskfile_id. However, it is clear that creating a new file requires two updates: updating the directory and creating the new file, which includes writing the i-node information. By enclosing these steps into one ARU, we turn the separate actions into one single atomic action. Consequently, if anywhere during these steps a system failure occurs, the changes made by the partially completed ARU will be undone after the system starts up again, thus leaving the file system in a consistent state.

Although ARUs do not support full concurrency control, they do provide some kind of isolation. Each ARU has its own private copy of its uncommitted data, which is not visible from other streams. So, any update made within a running ARU is not accessible from within other streams until that running ARU has committed. The commit atomically turns the uncommitted data of the ARU into the committed data, replacing any previous committed versions of the updated blocks. As a result of the commit, all updates of the ARU immediately become visible to other streams. In database terminology, ARUs avoid *dirty reads* (reads of uncommitted data other than your own), but they do not have *repeatable reads*, because a running ARU x can see all the changes of other ARUs that have committed while ARU x has been running. This means that two reads of a data block in ARU x will yield different answers if another ARU that has changed that same block has committed in between the two reads of ARU x.

Note that at any moment in time, there is only one committed version of each block, but there can be multiple uncommitted versions, depending on the number of running ARUs that have modified that block. However, in the current design of LD, using multiple ARUs to create multiple uncommitted versions of the same block is not recommended due to the lack of support for full concurrency control. This situation will likely not result in a serializable execution of the commands in the ARUs, unless the client giving the commands uses some external concurrency control method, enforcing the serializable execution.

As an example where LD does not provide concurrency control, consider the following example. Suppose that two clients (A and B) want to change the value of an integer which is stored in a block on disk. Client A increments the integer by 1, and client B decrements the integer by 2. Both clients do the following actions, but each in its own stream:

(1) `ld_start_aru(...)` : start an ARU

(2) `ld_read_data(...)` : read block from disk into memory

(3) Client A: add 1 to the integer, which is located in the block
Client B: subtract 2 from the integer, which is located in the block

(4) `ld_write_data(...)` : write the block with the updated integer back to disk

(5) `ld_commit_aru(...)` : commit the ARU

Concurrent execution of these two ARUs in two streams may lead to a race condition. For example, Figure 3.2 shows such a nonserializable execution of the commands sent to

LD, which leads to the *lost-update anomaly*. The problem is that client B reads the block with the integer before client A has committed its ARU, which means that B updates the integer based on its original value 101. As a result, the integer update of A is overwritten when B commits its ARU, and consequently, the new value of the integer is 99 instead of the expected 100. A serializable execution of these commands can only be guaranteed with proper concurrency control mechanisms.

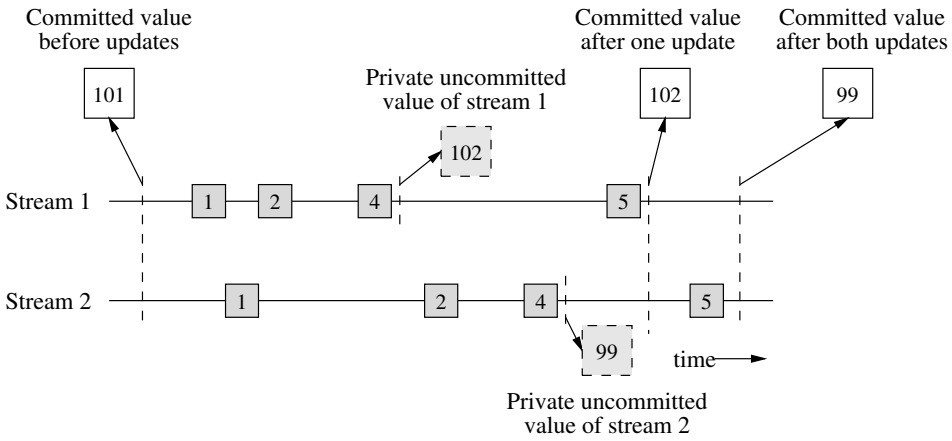


Figure 3.2: Lost-update anomaly. 1) Start ARU. 2) Read block. 4) Write block. 5) Commit ARU. Note: step 3 is not shown in this figure.

In the current design of LD the commit of an ARU always succeeds. However, this behavior may change in the future when ARUs are extended to support full transaction semantics. To support such transactions LD needs to include algorithms to support concurrency control and to prevent or resolve deadlock situations. Depending on the algorithms chosen, it may be possible that LD needs to abort a transaction in certain situations, which discards all the tentative changes made by the transaction. For instance, a reason for this could be that LD needs to resolve a deadlock situation in which that transaction participates.

3.6 Read-Ahead

A mechanism to support read-ahead is another requirement described in the previous chapter (Requirement 4, on page 42). In short, the motivation for support of read-ahead in LD is that effective read-ahead requires information on both the logical structure of the data, as well as the physical layout of the blocks on disk. The client has the knowledge about the logical structure, but only LD has access to the physical layout of the data. Therefore, effective read-ahead is only possible if the client and LD work together since the necessary knowledge is divided between the two. This complexity is a consequence of separating file management and disk management and adding location transparency.

```
/* Read a given number of blocks from a given position in a given disk file,
 * and try to read-ahead more blocks up to the specified maximum.
 * The supplied buffer must be big enough to hold the specified maximum number
 * of blocks. On success, the actual number of read blocks is returned. */
int ld_readah_data(uint32_t stream, uint32_t aru,
                   uint32_t cluster, uint32_t file, uint32_t offset,
                   uint32_t min_count, uint32_t max_count, char *buf);
```

Listing 3.6: Read ahead data blocks.

In LD we use an approach that allows the client to indicate to LD that read-ahead is desirable, but leaves the decision of exactly how much to read-ahead to LD. LD can then decide based on the physical layout of the data whether it will read-ahead, and if so, how much. For the support of read-ahead, LD provides the client with a function call by which the client can indicate the requested read-ahead. Furthermore, LD has a buffer of main memory, which serves as a **cache** for the data that has been read from disk.

When a client wants to read data and also wants to indicate that read-ahead is desirable, it calls the function `ld_readah_data`. Besides the arguments to specify from which disk file to read, the starting position in that disk file and a buffer in which LD can store the data, this function takes two more arguments: the minimum amount of data that the client expects to receive and a maximum amount that the client is willing and prepared to accept. This function is given in Listing 3.6. The minimum amount is the amount of data that LD must at least read from disk and return to the client. The maximum amount indicates that LD may return more data, up to the specified maximum. How much data LD actually returns beyond the specified minimum is up to LD and depends on the amount of overhead required to return that extra data. The overhead is determined by the physical layout of the data. If one or more expensive seeks are required to read the extra data, LD will not return it. However, if the extra data is well clustered with the data that needs to be read as part of the minimum, then it is relatively cheap to read and return the extra data up to the point that an expensive seek is required or the maximum has been read.

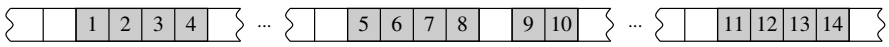


Figure 3.3: Read ahead in LD.

As an example look at Figure 3.3. In the figure the physical blocks of a disk are represented as an array of which only three parts are displayed. Suppose that the blocks that are marked gray belong to a particular disk file. We have marked each of these blocks with the block offset within that disk file. Notice that the physical blocks are not clustered around one spot, but are spread over the disk. Now imagine that a client wants to read the disk file. It can use the read-ahead function of LD to indicate that it wants to read at least eight logical blocks of that disk file, but would like LD to read-ahead more logical blocks of that disk file if possible, with a maximum of all 14 logical blocks. As ordered, LD will read at least logical blocks 1 through 8. This read requires two seeks: one to go to the

beginning of the file and one to bridge the big gap between logical blocks 4 and 5. To read more blocks, LD needs to skip one physical block on disk. The cost to read the extra two blocks is relatively small, so LD may decide to read these blocks as read-ahead. It is even likely that to skip one block, no actual seek is required. Even if it does require a head switch or even a one-track-seek, the benefits of the read-ahead may outweigh the cost of a switch or seek. On the other hand, LD may decide that such a small cost is too expensive already. To read logical block 11 would, however, certainly require a larger seek to reach it, so LD will decide not do that, and return only 10 blocks out of the maximum.

After returning the data, LD may immediately continue to read logical blocks 11 through 14 in the background if it has time to do so, for example, if the disk is idle. The philosophy behind this approach is that LD anticipates that the client will return with a request for these blocks soon. By first returning the 10 logical blocks, the client can immediately start processing the data it has received from LD, and by the time the client issues the request to read the other blocks, LD may already have those blocks in its cache. The choice whether LD will read the blocks ahead in background depends on many factors, including disk traffic and the available buffer space in the cache of LD.

Chapter 4

Architectural Overview

The previous chapter presented the main abstractions provided by LD, such as disk files, disk clusters, ARUs and streams. The following five chapters (Chapters 4 – 8) take a detailed look inside LD and present the internal structures of LD that are necessary to implement the abstractions from the previous chapter. Some requirements from Chapter 2 have not been dealt with in Chapter 3. In the following chapters, we will also present how LD fulfills these remaining requirements.

The discussion of the internal structures of LD has been split into multiple chapters. We start by presenting an overview of the internal organization of LD. This first chapter introduces the major components of LD. First, we look at the different types of data that LD stores on disk and how LD uses these types of data. This discussion is followed by a number of examples that illustrate the inner working of LD. These examples give a brief overview of LD as a whole and provide some insight in the internals of LD.

Chapters 5 – 8 are devoted to a more thorough discussion of the internal data structures of LD. In each chapter, we focus on one particular type of data, and present the data structures and algorithms used to manipulate that type of data. This discussion starts in Chapter 5 by focusing on LD's log, which plays a major role in crash recovery. Chapter 6 discusses LD's metadata, which LD maintains to keep track of the data that clients store on disk with the help of LD. Chapter 7 deals with checkpoints, which LD needs to recover to a consistent state after a crash. The last chapter in this discussion, Chapter 8, focuses on the client data and where they are stored on disk.

4.1 Different Types of Data on Disk

LD distinguishes among a number of types of data. In this section, we briefly review these types and clarify their purpose. Each type of data is stored on disk. For this purpose, LD divides the disk in a number of logical **areas**. The actual physical locations and sizes of some of these areas on disk may vary over time to follow changing resource demands. Each area is named after the type of data it holds.

LD has four main types of data. We have already mentioned two of these types before: client data and metadata. Before we present the complete list, we need to clearly define

what we mean by client data and metadata, to prevent misunderstandings in the future.

We can distinguish multiple levels on which data play a role. Figure 4.1 shows four such levels. On top are the user applications. A user application uses the underlying file system (a client of LD, or *LD-client*) to store its data within files on disk. The file system, in its turn, uses LD to store its files within LD's disk files. At the bottom is the actual physical disk on which LD stores its disk files.

For completeness, we could have drawn another level on top of the user application layer, and called it 'users', representing human beings behind a computer. These human beings use the user applications, such as a word processor, to store data. However, the difference between users and user applications is not relevant to our discussion of LD. We are mainly interested in LD and the layers directly above and below it. It is, therefore, unnecessary to be able to distinguish details within the input to a file system or DBMS; using one user application layer in the figure is sufficient.

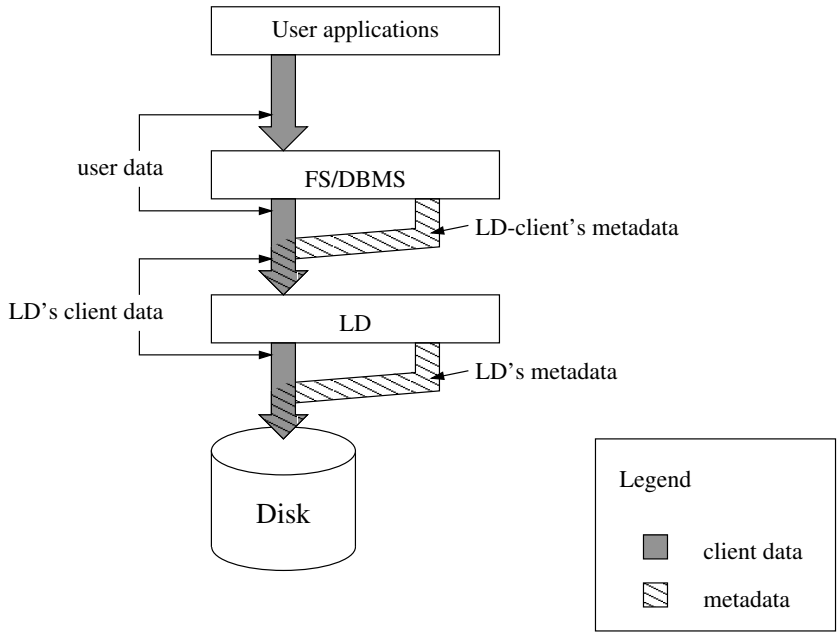


Figure 4.1: Data and metadata are present on different levels.

Each layer is responsible for storing the data it receives from the layer above it. Notice that the word 'client' is a relative concept. To LD, a file system is a client. However, to a file system, the application on top of it is a client; likewise, a human being can be seen as a client to the application. Therefore, on each level, the term **client data** can be used to refer to the data that are passed from one level to the next.

In order to keep track of the client data that the next layer receives, that layer also generates and keeps **metadata**. These metadata are passed, together with the client data

that a level received from the layer above, to the layer below. For example, in order to store the data of a user application, the file system generates its own metadata. *Both* the application's data (the client data of the file system) and the file system's metadata (e.g., i-nodes) are then passed to LD in order to be stored in disk files. LD does not make a distinction between the file system's client data and the file system's metadata that it receives. Seen from the viewpoint of LD, the combination of both is just a single input stream to LD originating from one of its clients; therefore, we call these data **LD's client data**.

In Figure 4.1 the passing of data from one layer to the next is graphically represented by arrows. The file system and LD level receive a data stream from the layer above. This stream is passed unchanged to the layer below, however, they also generate their own metadata stream. The next layer receives both streams, and views them as client data, which is represented in the figure by merging the data streams before they enter the next level.

The terms 'client data' and 'metadata' on their own are, therefore, ambiguous; their interpretation depends on the level on which they are used. In order to avoid confusion we will associate clear names to each of the client data and metadata streams on each level. These names are also given in Figure 4.1. The data provided to the file system level by user applications are called **user data**. Since user data are also the input to the file system, which is a client of LD, we also refer to user data as **LD-client's client data** or simply **file system's client data**, when it is clear that the LD-client is a file system. The file system passes user data through to LD together with it's own metadata, called **LD-client's metadata** or **file system's metadata**. LD receives its data stream from the file system as the combination of the file system's client data and the file system's metadata, and we refer to it as **LD's client data**. LD passes LD's client data through to the disk together with the metadata that LD generates, which is called **LD's metadata**. Throughout the rest of this dissertation we will often use the shorter terms client data and metadata to refer to LD's client data and LD's metadata, respectively.

Now, we can return to our discussion of the different types of data on disk. The complete list is as follows:

(1) **Client data**: the data that clients of LD, such as a file system or DBMS, store in logical blocks of LD. These data are mainly (see Section 4.1.1) stored in the **storage area** which is the largest area on disk. The layout of client data blocks in this area is important since clustering blocks leads to better sequential read performance.

(2) **Metadata**: the data representing administrative information needed by LD to maintain the blocks on disk and to support the abstractions of LD, such as disk files and disk clusters. The information includes data structures such as the *Mapping*, which is used to translate logical block addresses into physical block addresses, and the *FreeMap*, which keeps track of which blocks on disk are free and which are used. Metadata are stored in the **metadata area**.

(3) **Log data**: the data that LD stores in its **log**, representing a history of changes made to the disk by client commands. Each client operation that updates the state of the disk is logged at some moment. LD uses the log, among other things, to enable

it to recover after a crash. After a crash LD can recover to a recent consistent state with the help of a checkpoint (see next item) and the log. In short, LD first restores the latest successful checkpoint, which represents a consistent state of the disk, and then LD examines its log and uses the information therein to replay recent operations to recover to a more recent consistent state. Furthermore, the log is also used to support *collective writes* and to help implement the *no in-place update* policy. Log data and client data can refer to the same data blocks since client data can be (temporarily) stored in the log. The log is stored in the **log area**.

(4) **Checkpoint data**: the data representing a snapshot of a consistent state of the data on disk, that is, a consistent state of LD's client data and LD's metadata. LD regularly makes such a snapshot, which we call a **checkpoint**. A checkpoint is written such that the state represented by the checkpoint can be easily restored after a crash. Checkpoints enable LD to prune the log, which then only has to contain a history of update operations since the latest successful checkpoint. Therefore, the use of checkpoints speeds up the recovery process after a crash. Checkpoints are written in the **checkpoint area**.

Actually, there is one more, very small, area on disk: the **superblock area**. This small area is at the beginning of the disk, and is used by LD to start using the disk. The information stored in this area is similar to the information stored in the superblock of a file system. For instance, this information includes information about the location and size of each of the other areas on disk. We briefly discuss this area in Section 4.1.4, and will also come back to the superblock in Chapter 7.

Figure 4.2: Logical representation of the areas on disk.

A logical representation of all areas is depicted in Figure 4.2. Using these data types and data areas, we can explain LD in a nutshell. The storage area is used for storing LD's client data. The exact locations of LD's client data blocks in this area are monitored by LD to make sure that the clustering wishes of clients are obeyed as much as possible. Data blocks that are written by clients are usually first stored in the log, where they are collectively written in large segments. However, there are cases, such as large sequential writes of LD's client data, where client data may bypass the log. The exceptions on the basic rule that client data are first stored in the log will be discussed later on in Chapter 5. A separate cleaner process copies client data blocks out of the log into the storage area at a later point in time.

The log contains a history of all update operations issued by clients to LD. This history is used after a crash to help LD recover to a recent consistent state. To avoid having to store and replay all update operations since LD started using the disk, LD regularly makes

a *checkpoint*, which represents a snapshot of a consistent state of the data on disk. This checkpoint can be used as a starting point for recovery after a crash. As a consequence, checkpoints also enable LD to prune the history of update operations in the log by discarding all history before the latest successful checkpoint. The information that is used to create a checkpoint is in the metadata area.

Below we look at each of the different types and each of the corresponding disk areas in some more detail.

4.1.1 Client Data

Client data (short for LD's client data) refer to the data owned by the clients of LD. The majority of all client data are stored in logical blocks. Some client data can be stored in disk file headers, but, in normal use, this concerns only a relatively small amount of client data. We have mentioned two applications of these disk file headers in the previous chapter: for implementing immediate files or for storing a client's metadata. The logical blocks filled with client data are mostly stored on disk in the storage area. Only a small part of client data may reside in the log area and/or the metadata area. Not surprisingly, the storage area occupies the majority of the total storage surface of the disk.

Data blocks are grouped into larger logical units, namely disk files and disk clusters, which were discussed in the previous chapter. On request the data blocks of a disk file or disk cluster will be stored physically close together on disk (the clustering requirement). In other words, the layout of the blocks in the storage area is important. A reorganizer process is responsible for maintaining the physical clustering properties of the blocks in this area. We will come back to reorganizers in Chapter 8.

4.1.2 Metadata

The metadata area is the place on disk where LD stores its own metadata. LD's metadata consist of internal data structures that keep track of the logical blocks, disk files, and disk clusters of clients. Recall that LD sees LD-client's metadata, such as the i-nodes of file systems, as normal client data. If a client stores its metadata in logical blocks, LD cannot, and will not, see or make any distinction between LD-client's client data (also called user data) and LD-client's metadata. Such logical blocks are all stored in the storage area. The only way for clients to store their client data or their metadata in LD's metadata area is to use LD's disk file or disk cluster headers, which are stored together with LD's metadata in the metadata area. The reason for clients wanting to store their metadata near LD's metadata may be efficiency: storing data close together on disk may speed up access to them. However, LD's headers are only suitable for storing small amounts of client data.

The bulk of the client data is stored in disk files. LD's metadata are stored separately from such client data. The reason for this separation is that LD's metadata have different characteristics than LD's client data. In fact, both LD-client's metadata and LD's metadata have characteristics different from user data. For instance, they have different access patterns. This argument is similar to the argument we used for introducing disk file headers an disk cluster headers in Section 3.2.3. User data are often accessed sequentially, and therefore, effective physical clustering of that data is important. LD-client's metadata and

LD's metadata, however, are seldom read sequentially in large amounts. Additionally, caching can be used to successfully exploit any locality of reference in the access patterns to LD-client's metadata or LD's metadata, so that actual disk I/O to read that metadata from disk can be reduced. Therefore, LD has two separate storage areas: one for user data and another for LD's own metadata, and possibly also for LD-client's metadata. Note, LD does not force its clients to store their metadata together with LD's metadata, but only offers them the ability to do so, if they so desire.

The two main internal data structures for LD's metadata are the Mapping and the FreeMap. The **Mapping** is used to translate logical block addresses into actual physical disk block addresses, and to keep track of the logical relationships between blocks in disk files and disk clusters. The Mapping is logically seen a table of (key, value)-pairs, which in this case are (logical address, physical address)-pairs. The **FreeMap** keeps track of which physical disk blocks are in use and which are free. The FreeMap manages the entire disk, that is the blocks of all areas, holding both LD's client data as well as LD's metadata. The Mapping and the FreeMap are stored persistently on disk in blocks of the metadata area. For the understanding of the overview of LD we will briefly explain some details about the Mapping. A more thorough discussion of the internal design and implementation of both data structures is given in Sections 6.1 and 6.3.

Committed and Uncommitted Data in the Mapping

The Mapping stores the physical addresses of both committed and uncommitted client data blocks. Recall that uncommitted client data blocks are data blocks that have been written or deleted as part of a running ARU. Information about these uncommitted data blocks must be recorded in the Mapping for two reasons. First, these uncommitted data blocks are visible to commands within the stream and ARU that created these blocks. Second, the commit of that ARU will make these uncommitted blocks the new committed blocks, replacing the previously existing committed versions of those blocks. Although a logical block has only one logical address, it may have multiple physical instances, each with a different physical address: one for the committed and one for each uncommitted version of that logical block. However, this situation is only temporary; it lasts as long as there are ARUs running with uncommitted data. After these ARUs have been committed, there is only one instance of each existing logical block: the committed version.

How is the information about committed and uncommitted blocks stored in the Mapping? The Mapping must associate multiple physical addresses with one logical address, and still be able to know which block is the committed version, and which ARUs have created the other uncommitted versions. Therefore, LD uses an extended logical block address internally. Within LD the logical address that a client uses is extended with an extra field: the aru_id. An **internal logical block address** is thus a quadruplet: (aru_id, cluster_id, diskfile_id, offset). Committed logical blocks use an aru_id of 0, uncommitted logical blocks use the aru_id of the ARU in which they were created. This way, an internal logical block address uniquely refers to one version of a logical block, which has only one physical address associated with it. If a logical block has multiple versions (one committed and one or more uncommitted), then each uncommitted version has a different internal logical block address and has a different physical block address associated with

it. These pairs of internal logical block addresses and physical addresses are stored in the Mapping.

Committing an ARU in the Mapping

When an ARU is committed, the Mapping must be updated accordingly. The uncommitted logical blocks that have been changed within the ARU must now become the committed blocks. Logically seen, this process consists of the following five steps.

(1) Find out which blocks have been changed in this ARU, that is, find the uncommitted blocks of this ARU. All references to these uncommitted blocks can be found in the Mapping. These can be recognized in the Mapping since their internal logical block addresses start with the aru_id of the committing ARU.

(2) Find in the Mapping the entries of existing committed blocks that have been overwritten by uncommitted blocks in the committing ARU. These entries can also be easily found in the Mapping since their internal logical block addresses are the same as the internal logical block addresses of the uncommitted blocks of this ARU, except that the aru_id part is 0.

(3) Remove the entries of these previously committed blocks from the Mapping.

(4) Insert new entries for the newly committed blocks into the Mapping. These can be derived from the entries identified in the first step by setting the aru_ids of their internal logical block addresses to 0.

(5) Clean up the old ARU entries by removing the entries identified in the first step, which refer to uncommitted blocks.

Figure 4.3 illustrates how the Mapping is updated during a commit of an ARU. Figure 4.3(a) shows a small and simplified part of the Mapping of a hypothetical disk state. The Mapping holds the addresses of five logical blocks. The entries in the Mapping are kept sorted based on their internal logical block addresses. Three blocks belong to disk file 8 in cluster 3, and the other two belong to disk file 9 in cluster 3. The blocks are all committed versions, indicated by the aru_id 0 in their internal logical block addresses in the Mapping. The physical block addresses of logical blocks are shown in italics or bold italics. We use this convention throughout this dissertation. Now, suppose that the second block of disk file 8 is overwritten within a running ARU with aru_id 5. This new uncommitted block is physically written to disk, usually in the log. A new entry is written into the Mapping to refer to this uncommitted block. The new entry is shown in bold in Figure 4.3(b). The internal logical block address of this entry has aru_id 5 so that it is different from the committed version. Until this ARU is committed, there are two versions of logical block (3,8,2): a committed version, physically located at disk address *2331*, and an uncommitted version, physically located at disk address *66*. The uncommitted version is only visible to the stream that started ARU 5. When this ARU commits, the following happens. First, all entries in the Mapping that belong to this ARU and refer to uncommitted blocks of the committing ARU are found. In this case, the entry for logical block

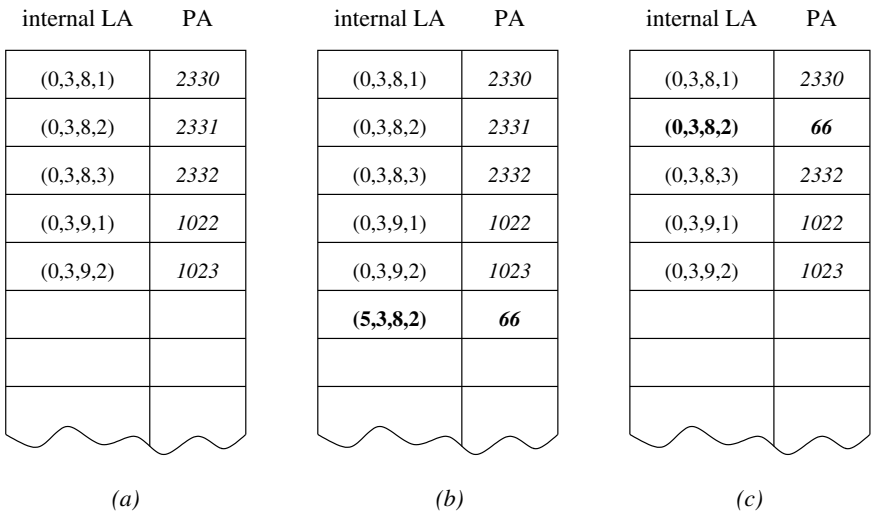


Figure 4.3: Effect of an ARU-commit on the Mapping. (a) Original mapping. (b) After writing block (3,8,2) in the ARU with aru_id 5. (c) After the commit of the ARU with aru_id 5.

(5,3,8,2) at physical address *66* is the only entry for this ARU. The second and third steps are to find the entry that refers to the old committed version of that block, and to remove this entry it from the Mapping, respectively. In this case, the entry for block (0,3,8,2) with physical address *2331* is found and removed. Fourth, a new entry for the new committed version is inserted into the Mapping. This entry has logical block address (0,3,8,2) and its physical address is *66*. Last, the old entry for the uncommitted version is removed. The result after committing this ARU is shown in Figure 4.3(c). After this commit, only the new committed version is accessible to all streams; the uncommitted version vanishes.

Note that the above steps logically explain what happens to entries in the Mapping when an ARU is committed. However, the actual implementation of these steps may be different as long as the end result is the same. For example, removing and inserting entries in the Mapping in five steps may be inefficient. Therefore, instead of removing the entries of the old committed blocks and inserting entries for the new committed blocks, it may be more efficient to simply update the physical addresses of these entries with the new physical addresses of the uncommitted entries.

In our example, the newly committed block will still reside in the log, which temporarily breaks the clustering of the blocks of disk file 8 since the second block of the disk file is in the log while the other two blocks are in the storage area. At a later point in time a cleaner process will move this block out of the log into the storage area and try to restore the clustering property of disk file 8. It is not unlikely that the cleaner process will copy this block to its old physical position (disk address *2331*), if this position is still available at that time. If not, then the cleaner process may use a different strategy to try to fix the clustering as much as possible. We will return to cleaner and reorganizer strategies in Chapter 8.

Note that in our example we have restricted ourselves to blocks that are overwritten in an ARU. However, it is also possible to delete existing committed blocks within an ARU. The representation of such delete operations in the Mapping requires more explanation, and will be discussed in Chapter 6 where we explain the Mapping in detail.

4.1.3 Log Data

LD uses a log for which it has reserved space on disk: the log area. The data written in this area are referred to as log data. The log actually has three purposes. One important purpose of the log is to hold a history of update commands from clients sent to LD in order to be able to recover to a consistent state. Examples of such update commands are `ld_write_data`, `ld_set_fh`, `ld_delete_blocks`, and also `ld_create_file`. Each of these commands is logged by writing a **log tuple** in the log. This log tuple contains such information that the operation can be redone when necessary during recovery. This information includes the type of command, such as a block write, a write of a file header, or a disk file creation, and if applicable, a pointer to the actual data blocks on disk that have been written consecutively as part of the command. Client commands that do not update data on disk are not logged since they are not relevant during recovery to a consistent state. Examples of such operations are the read operations `ld_read_data` and `ld_get_fh`.

A second purpose of the log is to help implement collective writes (Requirement 8, on page 42). LD uses collective writes to write client data to disk efficiently, which was discussed in Section 2.4.2. With this technique many small writes are accumulated in main memory and are turned into one large, consecutive, disk write operation. This enables LD to write large segments to disk and, therefore, to make better use of the available disk bandwidth. The question is where does LD write this segment on disk? Note that LD already needs to write a log tuple in the log for each client command that updates the data on disk. Therefore, LD usually also writes the client data in the log as well, so that writing the log tuple and the corresponding client data only requires one disk seek. In other words, client data blocks are written to the log in segments, which LD calls **log segments**. Each log segment contains two types of data: log tuples and client data blocks. However, note that our choice to write client data blocks in the log is not a necessity. They could also have been written elsewhere on disk. In fact, later in this chapter we will see that LD, in certain cases, does not write client data blocks in the log, but writes them directly in the storage area for efficiency reasons.

A third, and last, purpose of the log is to help implement the *no in-place update* requirement (Requirement 5). This requirement states that LD may never update blocks in-place, neither client data, nor LD's own metadata. Each log segment is appended to the log, and therefore, never overwrites any existing data. Note that when LD moves client data blocks from the log into the storage area, LD may safely update blocks in-place. This in-place update is allowed because these new client data blocks are already safe on disk in the log; therefore, in the event of a crash during the in-place update, the contents of these blocks can be restored with help of the data in the log afterward. However, we will soon see that LD writes its own metadata, and in certain cases, also client data directly to an area on disk outside the log. Therefore, the log is not the only mechanism used to fulfill the no in-place update requirement. In the rest of this chapter, we will return to the topic

of avoiding in-place updates whenever we deal with a subject that involves writing data to disk, and verify that the no in-place update policy is respected.

Cleaning Data from the Log

The client data blocks written in the log area are only temporarily stored there. Their final place is in the storage area where the blocks are stored in accordance with the clustering wishes of clients as much as possible. A *cleaner* process is responsible for copying client data blocks out of log segments into the storage area and to implement the clustering wishes. However, in order to improve performance, LD should minimize the overhead of such reorganizations, especially the required disk I/O, as much as possible.

Above, we already briefly hinted that it is not always the best option to write client data blocks in log segments into the log area. In some cases, these client data blocks need to be reorganized in order to restore their clustering properties. This is the case when clustering requirements of a client concerning these blocks cause the cleaner process to move these client data blocks near other blocks of the same disk file or cluster in the storage area.

However, in other cases, such clustering reorganizations are unnecessary. For example, if these client data blocks already form a large consecutive range of well-clustered data blocks. In that unfortunate case, LD must still move them from the log into the storage area because LD does not allow client data blocks to remain in the log indefinitely. Leaving blocks in the log indefinitely is not possible because LD uses the log area as a cyclic data buffer, and therefore, LD needs a cleaner process to clean old log segments to make it possible to write new log segments in the log area. This last case can be prevented, if LD does not write such client data blocks in a log segment in the first place. This is exactly what LD does: in some cases, LD writes client data directly into the storage area, instead of in the log area.

There are a number of cases where LD can bypass the log for client data blocks, which will be discussed in Chapter 5. In those cases, LD may write client data blocks *directly* into the storage area with the collective write technique. Note that we still want to use the collective write technique to write blocks into the storage area segmentwise in order to utilize the available disk bandwidth. The locations where these segments are written within the storage area must be chosen such that it does not overwrite any existing data, thus avoiding in-place updates. We call writing a number of data blocks directly into the storage area, instead of the log area, writing a **direct segment**.

Client data blocks that are not written in a direct segment, are written in a log segment. LD accumulates enough of such client data blocks in main memory and writes them together in a log segment to disk. For data blocks in the log, reorganization is necessary afterward to restore their clustering property.

Until a cleaner process restores their clustering by copying the blocks out of the log into the storage area, the clustering requirements of those blocks are temporarily broken. However, LD has an internal buffer cache which holds recently read or written data blocks. If the size of the cache is large enough and the cleaner is reasonably quick with restoring the clustering, it is likely that read requests for these temporarily unclustered blocks can be satisfied from the cache, so that the read performance will not suffer at all.

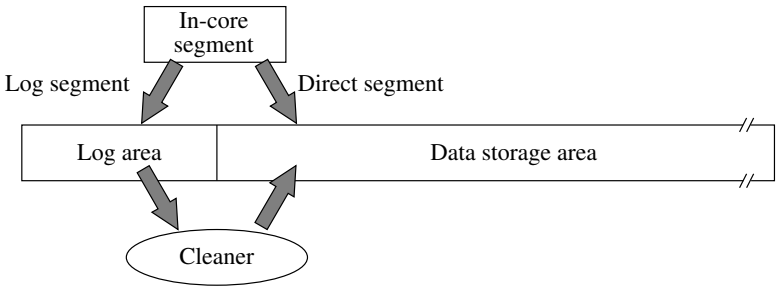


Figure 4.4: Collective writes in LD.

Figure 4.4 illustrates how LD writes data to disk. The figure only shows the log area and the storage area of the disk. The other areas have been left out for simplicity. New data blocks are first accumulated in main memory in an **in-core segment**. Based on some criteria, which will be discussed in Section 5.7, LD decides to write the in-core segment either in the log area as a *log segment* or directly in the storage area as a *direct segment*. In the background, a cleaner process moves data blocks that have been written in the log area into the storage area in order to comply with the clustering requirements. We will discuss this issue further in Chapter 5.

Note that, irrespective of whether the client data blocks are written in a log segment or directly into the storage area, the log tuples describing the client commands that have lead to these data blocks being written are always written in a log segment. This way, the history of client update commands is always in the log and can be used to recover to a consistent state. This makes the log the mechanism for LD to recover from a crash.

It may seem that bypassing the log only for client data blocks requires an extra seek to write them, compared to writing both the log tuples and the client data blocks in the same log segment. However, the overhead is smaller because writing client data directly into the storage area leaves room in the future log segment to hold other client data blocks. Therefore, it is possible that multiple direct segments are written, before the log segment containing their log tuples is written. That way the cost of an 'extra seek' is amortized over multiple direct segments. Furthermore, the advantage of not cleaning those client data blocks from the log is certainly worth the small overhead and some added code complexity.

A more detailed description of how the log, the log tuples, and recovery work follows in Chapter 5. In that chapter, we will also further discuss the advantages and disadvantages of writing direct segments and provide a categorization to help decide whether to write client data blocks within direct segments or within log segments. For now, it is sufficient to see the log as the place where client data blocks are collectively written in large log segments to achieve good write performance, and where log tuples are written to aid recovery.

4.1.4 Checkpoint Data

Periodically, LD makes a *checkpoint*, which represents a snapshot of a consistent state of the data blocks on disk, in the checkpoint area. This state can be easily restored after a crash. The reason for making checkpoints is twofold. First, a checkpoint allows the log, which contains log tuples to allow LD to replay the updates of clients after a crash, to be pruned. The log tuples whose updates have already been incorporated in the state represented by the latest successfully made checkpoint can be discarded from the log. This allows LD to shrink the log, freeing space in the log area.

The second reason is to speed up the recovery process. During recovery, LD restores the most recent checkpoint in the checkpoint area containing a recent consistent state, and then uses the log to redo any updates that have been issued by clients after that checkpoint had been made. By regularly making a checkpoint, the number of updates that has to be redone using a log tuple in the log is small and this decreases the time needed for recovery. However, making a checkpoint during normal operation of LD also takes time and uses resources, so a compromise between the overhead of making checkpoints and recovery time must be found.

The size of the checkpoint area on disk is chosen such that it can hold multiple checkpoints simultaneously. This is necessary because LD may not overwrite the previous checkpoint when making a new checkpoint; this would violate our no in-place update policy. How does LD, after a crash, know where the latest successful checkpoint is? The answer is that the location of the checkpoint area on disk is stored in the superblock.

To guarantee the integrity of the superblock, LD also does not overwrite the superblock in place. Similar to the checkpoint area, the superblock area can hold multiple superblocks simultaneously. These superblocks are stored at predefined locations on disk (e.g., at the beginning of the disk), which LD uses alternatingly to hold the latest version of the superblock. The latest version of the superblock can be found by scanning all superblocks and examining the sequence number included in each superblock.

After LD finds the superblock, it can then find the checkpoint area. During recovery LD can then scan this checkpoint area to find all checkpoints contained in them and determine which of them was successfully written most recently by examining a sequence number that is part of each checkpoint.

Even though, roughly stated, the consistent state of the data on disk consists of very much data, namely all client data blocks and metadata blocks that are valid at the time of making the checkpoint, LD only needs to write relatively very little data in the checkpoint area itself to make a checkpoint. This way LD can make checkpoints efficiently. How exactly LD makes its checkpoints, including other issues, such as how to guarantee that the state represented by the checkpoint is consistent, what to do with running ARUs, and how recovery works, is treated in Chapter 7.

4.2 Examples of Typical Operations

The previous section has given a brief introduction into the basics of how LD works internally. This section illustrates these basics using some typical commands that clients send to LD. We look at the following operations that a client can initiate:

(1) Read a single data block from disk via a simple ARU.

(2) Write a single data block to disk via a simple ARU.

(3) Write a single data block to disk via a composite ARU.

(4) Create a new disk file via a simple ARU.

(5) Write file header data via a simple ARU.

Each of these operations is clarified with the use of a diagram showing the major components of LD. Most of these components are data structures which have already been introduced before, such as the *Mapping* and the *FreeMap*. There are three new components in the diagram. We will briefly introduce them here. The first one is the *Cache*. In a previous chapter we have already hinted toward a cache in LD. The cache is an amount of buffer space that can hold data blocks in main memory. The main purpose is to hold currently or recently used data blocks. In our diagram the cache component represents a component that also takes care of the actual physical reading and writing of blocks from and to disk.

The second new component is the *Stream/ARU administration*. This component keeps track of which streams have been created by clients, and which ARUs are running. LD needs to know this information to monitor the integrity of each user command. For example, LD forbids multiple ARUs running within one stream at the same time, and only allows commands to be sent on existing streams.

The third component is the *In-core segment/Log tuples*, which is responsible for maintaining the in-core segment in which newly written data blocks are accumulated to help implement collective writes. As explained in a previous section, for each operation a log tuple is also written in the log for recovery purposes. To keep track of the data blocks in the in-core segment, this component contains a table of pointers to buffers in the cache that hold the data blocks that are in the in-core segment. The log tuples are also stored in one or more data blocks, which are also stored in the cache. We assume that this component is able to manage the in-core segment and generate the necessary log tuples, including allocating buffers in the cache to hold these log tuples. Furthermore, when the time comes, this component is responsible for writing the in-core segment to disk, either in the log area or the storage area, and writing the log tuples in a log segment.

Interactions between these components are represented by arrows. We use two kinds of arrows: thick arrows and thin arrows. The thick arrows represent the flow of client data from one component to another. For example, in a write command, a thick arrow would represent client data blocks that are to be written to disk. The thin arrows represent the flow of other types of data, such as logical addresses, or represent invocations of some action that is done within a component. For example, if the Mapping is needed to find the physical address of a logical block with a certain logical address, a thin arrow leads into the Mapping component, representing the logical address of the block. At the other end of the Mapping component, another thin arrow leaves the component, representing the corresponding physical address of the block. Another example is a thin arrow leading into the In-core segment/Log tuple component to represent the action that a log tuple is generated and stored in the in-core segment.

The figures are only a simplified representation of LD and contain just enough components and interactions between them to explain the examples in this section. The actual implementation contains more components and their interactions are more complicated. For example, one issue that is explicitly left out of our discussion of the examples is how the Mapping is stored on disk and when updates to the Mapping are made persistent on disk. Such issues will be discussed in Chapter 6.

Example 1: Reading a single data block from disk

The first example we look at is a read operation, which is initiated by a call of the function `ld_read_data`. We look at a simple ARU read, which should return the committed version of the requested block. Figure 4.5 shows a graphic representation of what happens inside LD during a read operation. Suppose that in this example the client issues a read request to read a single block at position 5 in disk file 8 in cluster 3. The request is issued in stream 1 and is sent as a simple ARU, so the supplied aru_id to the function call is 0.

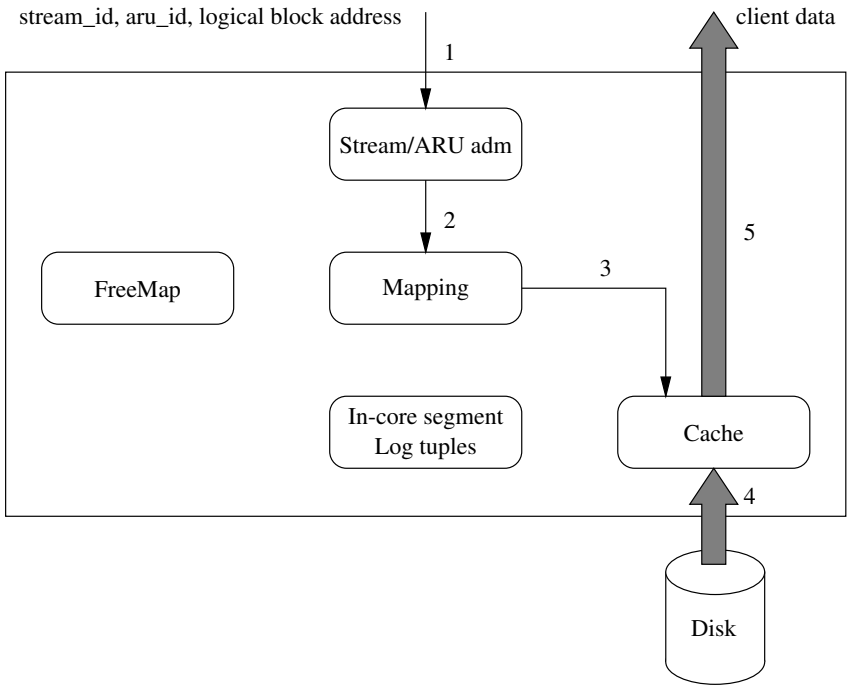


Figure 4.5: Execution of `ld_read_data`.

Now consider what happens inside LD while it serves this client request. The first thing that LD checks is whether the stream_id and aru_id are correct (step 1 in the figure). The stream must have been created and there may not be a running ARU on that stream, as this request has been sent outside an ARU. We assume that all is correct here.

Step 2 consists of translating the logical address of the requested client data block to the physical address where the block actually resides on disk. Recall that the Mapping uses an internal version of the logical block address. The internal logical block address is a quadruplet (aru_id, cluster_id, diskfile_id, offset), which in this case is (0,3,8,5). Since the read is done outside an ARU, the aru_id within this quadruplet is 0. This address is used to seek the corresponding physical address of the block in the Mapping. Suppose that the block exists on disk, and the corresponding entry in the Mapping reveals it has physical address *4125*.

In step 3, the block with physical address *4125* is requested from disk. However, the Cache component first checks if it has the requested block in one of its main-memory buffers. If so, it can return the requested data to the client immediately (step 5). Otherwise, an actual disk transfer must be initiated, as is done in step 4. The data will be read into a buffer of the cache, after which it can be returned to the client.

Example 2: Writing a single data block to disk

In this example we look at a write operation (`ld_write_data`) outside an ARU. Figure 4.6 shows how LD handles this operation internally. Again, we use the stream and logical address that we used in the previous example: stream 1, cluster 3, disk file 8, offset 5.

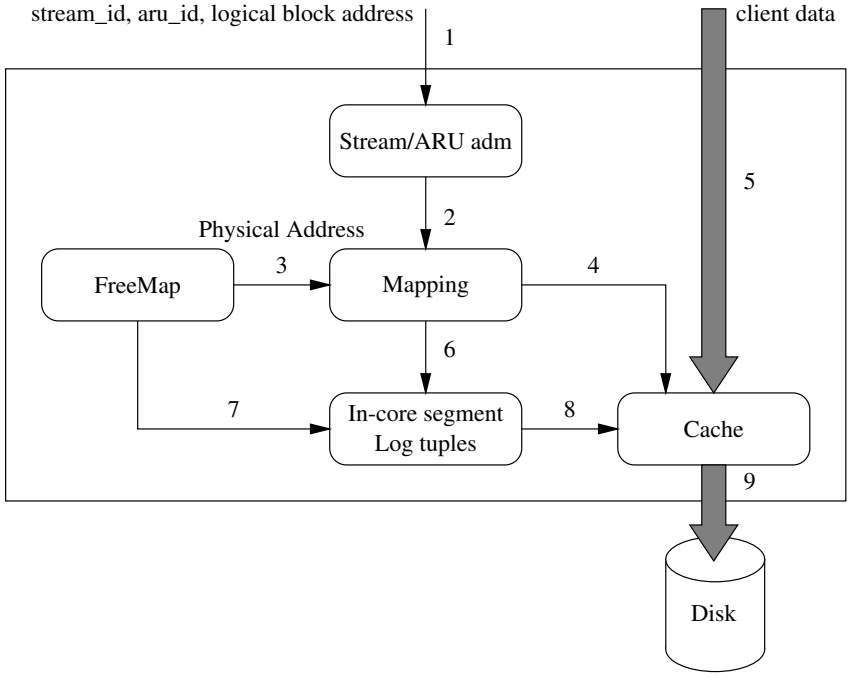


Figure 4.6: Execution of `ld_write_data`.

In step 1 LD checks the stream and aru_id as was done in our read example. In the following steps 2-5, LD stores the single client data block that must be written to disk, in a main-memory buffer of the cache and inserts an entry in the Mapping for this block. In step 2, the internal logical address of this block (0,3,8,5) is given to the Mapping. Next, LD must allocate a physical block address for the client data block. This block is first stored in the cache, and will stay in the cache until enough dirty data blocks have accumulated in the cache to write a large segment to disk. Until LD decides otherwise, the block is scheduled to be written in a log segment in the log area. However, LD may decide to write this block within a direct segment into the storage area instead. We will come back to this issue at the end of this example. For now, LD will assume that the block is written within a log segment.

In step 3 LD allocates a physical block address for this block. The figure shows that in step 3 the FreeMap is consulted. This may create the impression that a physical address is requested for each block in the log segment separately. This is not so: in practice, LD can optimize this step by allocating all necessary blocks for a complete log segment in one request to the FreeMap. When LD starts accumulating data blocks for a new log segment, it immediately allocates the space for a complete log segment (i.e., a range of consecutive blocks) in the log area. Then, whenever LD wants to add a new block to the log segment, it uses one of these pre-allocated blocks. The assignment of physical block addresses for a complete log segment is not difficult. The log area is simply filled with log segments one at a time in a round robin fashion. However, since LD can write log segments to disk that are not completely full, LD updates the FreeMap again when LD actually writes the log segment to disk by indicating which blocks in the log segment are still unused.

With the above mentioned optimization, assigning a physical address for block (3,8,5) in step 3 of our example, actually consists of only selecting an address from the range that LD has already allocated for the entire log segment. Suppose that LD assigns physical address *6872* to this block. The Mapping is then updated with an internal logical address and physical address pair, which in this case is the pair ((0,3,8,5), *6872*).

Steps 4 and 5 in the figure represent the action that the client data is stored in a buffer in the cache, and the address *6872* is associated with it in order for it to be found on future read requests.

Step 6 represents the action that the block is put in the in-core segment together with a corresponding log tuple. The in-core segment is filled with client data blocks, but also contains the log tuples that correspond to the operations that wrote the client data blocks in that in-core segment. These log tuples are small, and therefore, only a small number of blocks of the future log segment are needed to store these log tuples. Logically, the physical addresses of these log blocks are requested from the FreeMap in step 7, and the corresponding buffers are requested from the cache in step 8. However, when LD has already pre-allocated all blocks of the log segment on disk, as mentioned above, step 7 consists of only selecting the necessary blocks from this pre-allocated range of blocks. The in-core segment does not have an actual copy of the client data blocks, but only maintains a list of pointers to buffers in the cache to keep track of which blocks of the cache are part of the in-core segment.

Step 9 represents the flushing of the in-core segment to disk in the form of a log segment. This step happens when enough data has been accumulated in the in-core segment.

The actual disk I/O consists of writing the log segment including the client data blocks and the log tuples to disk in one large contiguous write.

In a previous section we discussed that LD could also write client data directly into the storage area instead of in a log segment in the log area. If LD decides that it is better to write the accumulated data blocks of the in-core segment into the storage area as a direct segment, LD consults the FreeMap to find new physical addresses in the storage area for these blocks. After making the necessary changes in the Mapping, the cache and the log tuples, LD can write these blocks to disk in the storage area. However, the log tuples, which are still in main memory, must be written to disk as part of a log segment. LD can either wait until enough client data blocks that have to be written in the log area have accumulated, and write the log tuples to disk as part of that log segment write, or LD can write a partial segment to disk immediately. The first option may often be preferable since it utilizes the bandwidth of the disk more effectively. We will discuss direct segments in more detail in Chapter 5.

Example 3: Writing a single data block via a composite ARU

This example explains how LD deals with a write in an ARU. It consists of three operations. First, an ARU is started, then a write is done within that ARU, and finally the ARU is committed. We will look at each of these operations in turn.

Operation 1: Starting an ARU

A client starts an ARU by calling the function `ld_start_aru`. This function sets up the ARU so that it can be used in future calls to LD. Figure 4.7 shows how this is done. In step 1 LD checks in the ARU administration whether an ARU can be started in that particular stream. If another ARU is already running in that stream, an error is returned to the client. Otherwise, a free aru_id is allocated, and LD registers that this stream has started an ARU.

Furthermore, a log tuple is generated stating that an ARU has started. This log tuple is stored in the in-core segment (step 2), and if necessary, a new block in the log segment is allocated to hold the log tuple (steps 3 and 4). The aru_id that has been assigned to this ARU is returned to the client in step 5. In time, the log segment is written to disk, which is shown by step 6.

Operation 2: Writing a single data block in an ARU

The process of writing data blocks within a running ARU is almost the same as writing data blocks outside an ARU, which we discussed in the previous example. The only difference is that the aru_id supplied to the function `ld_write_data` is nonzero. As a consequence, the internal logical block address that is used in the Mapping will not overwrite the entry for the committed version of that block, but will create a new entry. During the commit of the ARU this entry will replace the entry for the committed block, which will be discussed next.

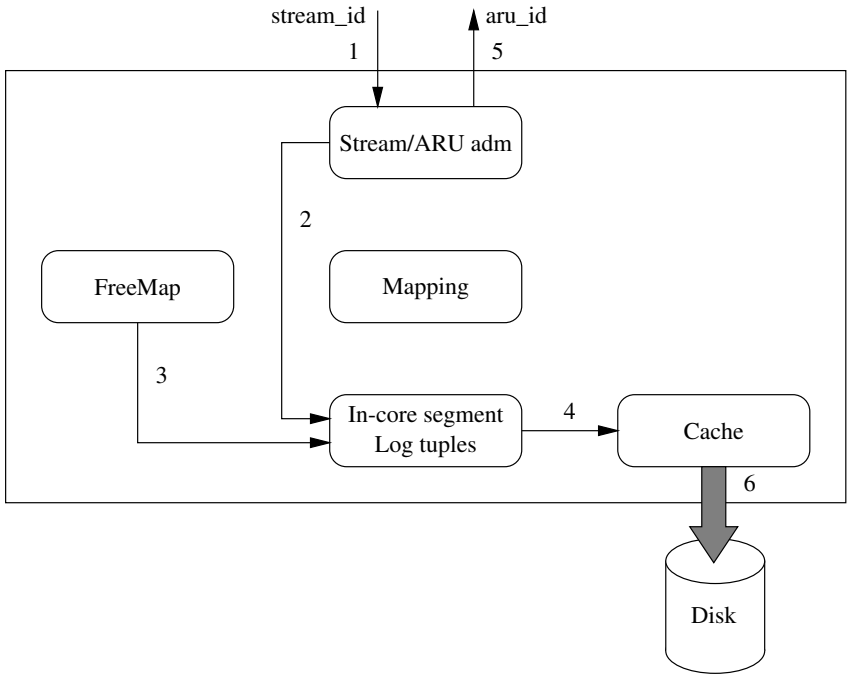


Figure 4.7: Execution of `ld_start_aru`.

Operation 3: Committing an ARU

The commit of an ARU signifies the end of a group of operations that should be recoverable as a single unit. Figure 4.8 shows how this is done. The operation starts the same as the other commands: the ARU administration is consulted to check whether the supplied aru_id and stream_id are valid. In this case the aru_id must correspond to the currently running ARU in this stream. At the end of this operation, the ARU administration is updated to reflect that there is no ARU running in this stream anymore.

Step 2 represents the action of updating the Mapping. The uncommitted changes that have been made as part of the ARU must now become committed changes. The required changes applied to the Mapping have already been discussed in Section 4.1.2. In short, this is done by removing the old entries that refer to the old committed blocks. Furthermore, for each entry in the Mapping that refers to uncommitted blocks of this ARU, recognizable by a nonzero aru_id in the internal logical block address, a new entry is inserted into the Mapping, which now refers to a committed block, that is, its aru_id in the internal logical block address is 0. Last, the old entries of the ARU are removed from the Mapping. This whole operation is done atomically, and as a consequence all changes made within the ARU become visible at the same time.

Steps 3, 4, and 5 refer to the log tuple that is written in the in-core segment denoting that a commit has been done. Step 6 represents the flushing of the log segment at which

time this log tuple is written to persistent storage.

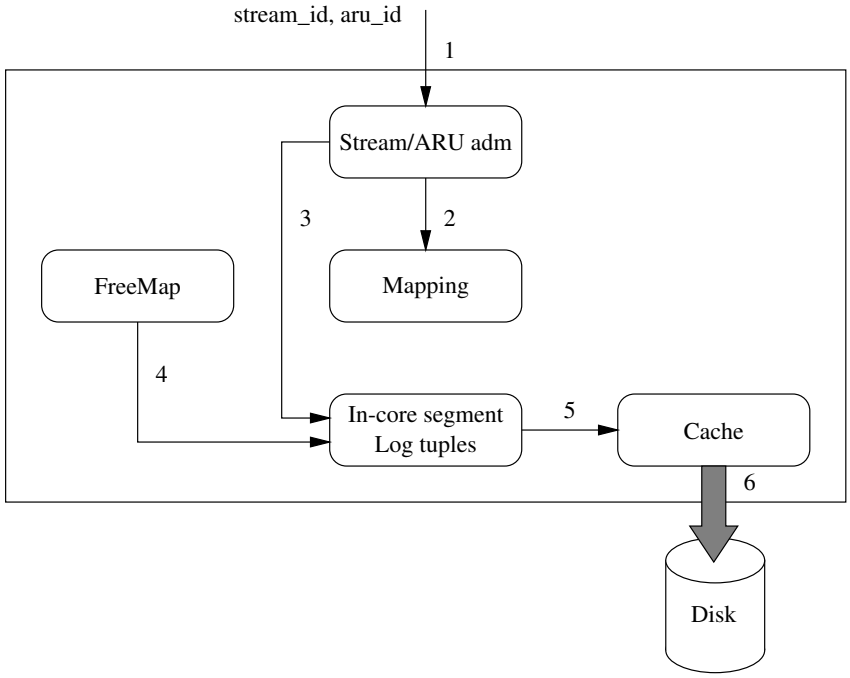


Figure 4.8: Execution of `ld_commit_aru`.

Example 4: Creating a disk file

In this example, we look at the creation of a new disk file. A new disk file is created with the function call `ld_create_file`. The internal administration of LD regarding the disk file and disk cluster hierarchy is also stored in the Mapping. Recall that, conceptually, the Mapping is a table of (key, value)-pairs. LD uses the presence of a special (key, value)-pair to indicate the existence of a disk file or disk cluster. We call this special pair a **header** entry. The reason for this name is that this header entry is also used to hold the file or cluster header associated with the disk file or disk cluster, which is illustrated in our next example.

The key of a header entry has the same format as an internal logical block address: a quadruplet (aru_id, cluster_id, diskfile_id, offset). For a header entry, the offset in this tuple is 0. This makes a header entry distinguishable from other (key, value)-pairs in the Mapping that hold physical block addresses because the latter have offsets > 0. Since a disk file cannot have more than one disk file header, reserving the offset 0 to identify the disk file header is sufficient. The cluster_id and diskfile_id form the unique identification of the disk file. The aru_id indicates whether the disk file has been created within a still

running ARU, that is, whether its creation has already been committed or not. A header entry for a disk cluster has the diskfile_id also set to 0. Since diskfile_id 0 is now reserved to refer to the cluster header, the first available diskfile_id for real disk files is 1. In short, a header entry can be distinguished from other entries in the Mapping by looking at the key. The key of a header entry has an offset of 0, while other entries that are used to correlate logical addresses and physical addresses have an offset greater than 0, because the first data block is at position 1 in a disk file. We will come back to the storage mechanism of the Mapping in Section 6.1, where we discuss the Mapping in greater detail.

The value field of the (key, value)-pair in the Mapping has a variable size. If the entry in the Mapping is used to map one logical address to one physical block address, the value field holds a physical block address, which is currently 4 bytes in size. The value field of a header entry, however, consists of a *fixed-size* and a *variable-size* part. The fixed-size part is used by LD to store LD's relevant (meta)information about the corresponding disk file or disk cluster. The variable-size part is used to hold any file header or cluster header information from clients (see our next example). We call the fixed-size part the **private part** of a header because only LD can access this part of the header. The variable-size part is called the **public part** of a header because a client may use this part to store its client data (i.e., user data) and/or its metadata (i.e., LD-client's metadata). When a disk file or disk cluster is first created, this public part of the value field is empty.

The size of the private part is currently only 1 byte, and contains a number of bit flags. Here we only mention one use of these bit flags since the others are more implementation specific. One bit is reserved to indicate whether this disk file or disk cluster needs physical clustering. The current version of LD stores only very little information in this fixed-size part. However, in future, LD may expand it to include more information. For example, LD could store the number of blocks allocated for a disk file in the header entry of that disk file, or the number of disk files within the header entry for a disk cluster, or even information concerning access control, or security more in general.

Now let us look at Figure 4.9, which illustrates how a disk file is created. The operation is started by calling `ld_create_file`. The client supplies the stream_id, aru_id, and the requested cluster_id and diskfile_id. Step 1 checks the integrity of the stream_id and aru_id parameters.

In step 2, LD first checks whether that disk file already exists. If so, an appropriate error is returned to the caller. Otherwise, a new header mapping entry is created in the Mapping to indicate that the disk file now exists. The value field of this header entry consists only of its fixed-size, private, part. Its public part, which is reserved to hold file header data of a client, is empty. Steps 3, 4, and 5 write the corresponding log tuple entry in the in-core segment, which is flushed at a later time, visualized in step 6.

Example 5: Writing a file header

Our last example illustrates writing a file header. Writing a file header is different from writing normal client data because file header data are stored as part of LD's own meta-data. More precisely, file header data are stored within the Mapping of LD. This way, reading file header data only requires accessing the Mapping. In contrast, reading data in logical blocks also requires reading the actual data blocks. Figure 4.10 shows this

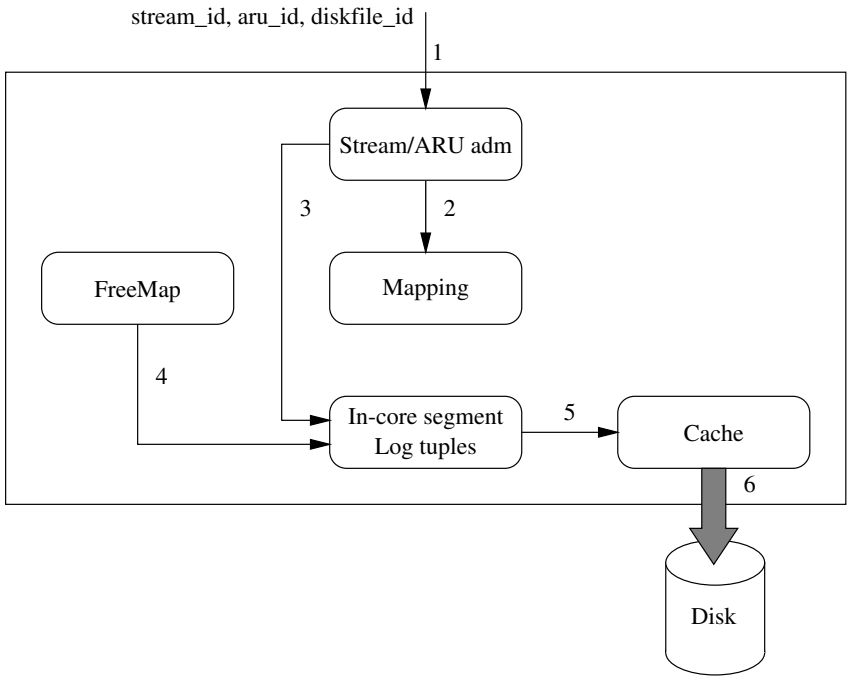


Figure 4.9: Execution of `ld_create_file`.

operation.

The previous example explained that every disk file and disk cluster has a special header entry in the Mapping. This (key, value)-pair has a value field, which consists of a private and a public part. The private part contains meta-information about the disk file or disk cluster, and the public part contains the actual client's header data.

Writing a file header involves the same actions as creating a disk file, which writes an empty file header. The operation is started by calling `ld_set_fh`. The client supplies the stream_id, aru_id, the diskfile_id whose file header is being written, and the data that are to be stored in the file header. Step 1 checks the integrity of the stream_id and aru_id parameters.

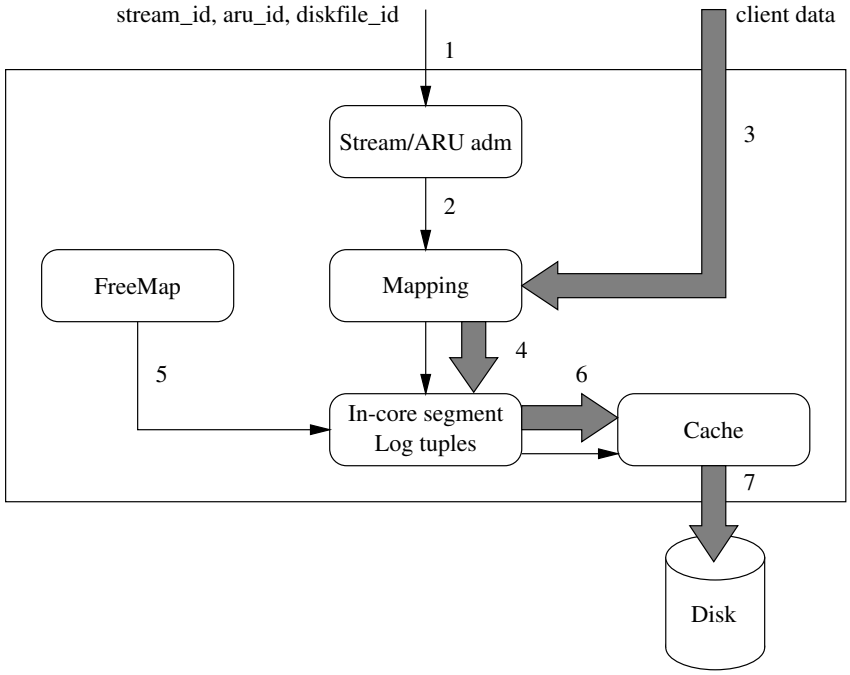


Figure 4.10: Execution of `ld_set_fh`.

In steps 2 and 3, the file header data supplied by the client are stored in the Mapping. First, LD checks whether the disk file exists by checking the existence of a corresponding header entry in the Mapping. When this entry is found, LD stores the file header data in the public part of the value field of the header entry. The contents of any previous file header data are replaced with the new data.

Steps 4, 5, and 6 represent a log tuple being generated, which is put in the in-core segment. Note that the file header data are also put in the in-core segment, otherwise the log tuple would not be complete, and could not be replayed in case of recovery. The last step, step 7, shows the flushing of the log segment to disk.

Chapter 5

The Log Area

The previous chapter has given an overview of the implementation of LD. LD organizes the disk into four areas, each containing a different type of data. Each type of data has a specific purpose in LD. The next four chapters focus on one specific type of data in turn. We start by taking a detailed look at the log data. In particular, we explain the structure and the function of the log in LD.

We have already mentioned that the purpose of the log is threefold. First, the log keeps a history of executed client commands. This enables LD to recover the disk to a recent consistent state after a crash. Second, the log helps to prevent in-place updates because data blocks that are written in the log are always written at unused disk locations. In particular, client data blocks that are written in the log do not directly overwrite blocks in the storage area. Third, the log helps LD to improve the write performance of disks by supporting collective writes. The first two purposes are related to data integrity; the last one is related to performance. Both the issue of improving data integrity as well as the issue of improving performance were identified as problem areas in Chapter 2.

In this chapter, we take a closer look at how the use of the log enables LD to achieve these goals. Sections 5.1 and 5.2 present the structure of the log itself by introducing the log area and log segments, which together make up the log. Next, we explain in Section 5.3 how LD stores the history of client commands in a log segment using log tuples. This is followed by a discussion of how LD forms and writes log segments. Before log data are physically written into the log area on disk as a log segment, LD accumulates the data in main memory in the in-core segment, which is discussed in Section 5.4. This gives LD the opportunity to optimize the process of writing log segments by reducing the amount of data written. This optimization is explained in Sections 5.5 through 5.7. Next, we discuss the conditions when LD writes a log segment to disk, and how LD can log a command using multiple log segments. We end this chapter by briefly looking at how LD cleans the log area.

5.1 Log Area

The log of LD is not like a traditional log in a database system. Such a log is typically separate from the actual database and often resides on a separate physical disk. The sole purpose of a database log is to provide integrity guarantees, especially with respect to the atomicity and durability properties of transactions. During normal operation, a database log is a write-only medium. The contents of the log are read only in the rare circumstance of recovery after a crash. LD's log, however, is an integral part of the storage system. All client data blocks that are written in the log are immediately accessible for normal read requests from clients.

LD stores its log in the log area. The **log area** has a fixed size and location on the disk. The size of the log area is currently set at 10% of the total disk capacity. The larger the log area is, the more space LD has to write entries in its log, and the longer it can wait before a cleaner process must clean the log area to create free space for new entries.

The log area is divided into a number of slots, called **log area slots**, each of which is 1 MB in size. Each slot can hold one **log segment**. Recall from Chapter 4 that a log segment holds the actual log data that is written to disk. Log segments are discussed in the next section. The slots of the log area are filled in a round robin fashion starting at the beginning of the log area with log segments. This allocation mechanism is simple and does not require an index structure on top of the log area to keep track of the log segments, which simplifies reading log segments during recovery.

5.2 Log Segment

As mentioned before, the log of LD is built up of **log segments**. Each log segment is written to disk in one large disk operation into a log area slot of the log area. The size of a log segment is chosen such that the overhead of seek and rotational delay is relatively low compared to the amount of data written. That way LD utilizes the available bandwidth of the disk effectively. Currently, the size of a log segment in LD is at most 1 MB. LD can write smaller log segments to disk. For example, LD may decide to flush a log segment prematurely to disk, and start filling a new empty in-core segment. This happens, for instance, when the data of a client write command does not fit in the available empty space of the current log segment anymore. The next log segment will be written in the log area in the next log area slot.

Starting log segments at well-known log area slot boundaries is a clear and simple design. However, writing smaller log segments in log area slots leads to temporarily having some unused space in the log area. Fortunately, disk space is not a scarce resource, and therefore, wasting a small percentage of disk space is not a problem. Note that LD does not waste main memory buffer space because it does not reserve 1 MB's worth of buffers for a log segment in advance, but only uses what it needs. Unfortunately, writing small log segments also leads to lower effective disk bandwidth utilization. Therefore, it is important that LD tries to write log segments that are as large as possible. We will come back to this issue in Section 5.8, where we discuss when LD writes log segments to disk.

The layout of a log segment is depicted in Figure 5.1. Each log segment starts with a **log segment header** and ends with a **log segment trailer**. Their purpose is to clearly

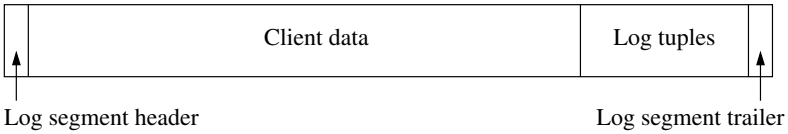


Figure 5.1: Layout of a log segment.

mark the begin and end of a log segment. They are needed to verify the integrity of the log segment. Since the log, which consists of log segments, plays an important role in the recovery process, LD must be able to rely on the integrity of each log segment. In our model, this dependency means that LD must be able to recognize whether a log segment has been successfully written to disk completely, or whether a crash in the middle of writing has resulted in only a part of the log segment reaching the disk.

Table 5.1: Contents of a log segment header.

Field	Description
log_segment_id	log segment sequence_nr
log_segblkcnt	total number of blocks in this log segment
log_clientblkcnt	number of client data blocks in this log segment

The contents of the log segment header and log segment trailer are given in Tables 5.1 and 5.2, respectively. The header contains a **log segment identifier** or log_segment_id, which is a unique 4-byte integer. We use a sequence number as log_segment_id, which is used by LD to find the last log segment that has been successfully written before a crash. The header also records the total number of blocks in the segment that are filled with data, and records the number of those blocks that contain client data. The remainder of the used blocks contain log tuples. The log segment trailer consists only of a checksum. The checksum currently consists of 32 bits and is calculated over all client data blocks, all log tuples and the header in the log segment.

Table 5.2: Contents of a log segment trailer.

Field	Description
log_chksum	checksum of the log segment

A log segment is written in one large sequential disk write operation. When reading a log segment from disk, LD checks its integrity by examining the information in the log segment header and trailer. The log segment header is stored in the first sector of the segment. The position of the trailer can be derived using the position of the header and the

size of the log segment, which is stored in the header. The integrity of the contents of the log segment is checked by calculating the checksum of the log segment which must match the checksum in the trailer. If this check fails, LD knows that the log segment has not been written completely. An incomplete log segment is considered not to have been written at all, and therefore, the changes recorded in that log segment will not be recovered. Additionally, the log_segment_id field in the header is used to determine whether this log segment is the successor to the previous log segment or whether it is an old, possibly obsolete, segment.

LD uses a 32-bit checksum. We expect this is enough to guarantee the integrity of a log segment under most circumstances. However, the checksum can easily be extended to a 64 or 128-bit checksum, increasing LD's ability of detecting errors at the cost of increasing the overhead of calculating the checksum.

Between the log segment header and log segment trailer are the client data blocks and the log tuples. Recall that the log contains the history of recently executed client commands that have updated data on disk. Each command is represented by one **log tuple**, which describes the type of command (e.g., a create or write command) and which logical blocks, disk files or disk clusters are concerned. In other words, each log tuple describes the command in such a way that it can be redone at a later point in time. We will look more closely at log tuples in Section 5.3.

The actual data blocks that client commands, such as `ld_write_data` and `ld_set_fh`, write are often stored in the client data section of the same log segment as the log tuple describing that command. Note that a command as `ld_write_data` writes logical blocks containing client data. However, a `ld_set_fh` command writes header data, which is also logged to disk in the client data section of a log segment. All client data blocks and headers in the client data section of the log segment are described by exactly one corresponding log tuple in the same log segment. However, note that a single log tuple can refer to multiple data blocks. For example, a single log tuple may describe a write command writing multiple consecutive client blocks.

Log tuples are stored at the end of the log segment. A log tuple has a fixed size, which makes a log tuple easy to manipulate. A log tuple that describes a write command with corresponding client data blocks, contains a pointer to these client data blocks. Often, these client data blocks are located in the same log segment as the log tuple. There is one exception, which occurs when the client data blocks are written directly into the storage area using *direct segments* (see Section 5.7). The client data blocks present in a direct segment bypass the log, but the corresponding log tuples are still written in a log segment, as mentioned in Section 4.1.3. In this case, the pointer in the log tuple points to the client data blocks that have been written in the storage area.

We have chosen to store client data blocks and log tuples separately in the log segment. The reason for this separation is efficiency. Recall that the smallest unit that a disk can access is a sector. A log segment is stored on disk in a consecutive range of disk sectors. A logical block fits exactly in one sector; therefore, it is efficient to store client data blocks in the log segment aligned on sector boundaries. This way, these data blocks in the log segment are immediately accessible. On the other hand, log tuples are small. Therefore, for space efficiency and to avoid large internal fragmentation in the log segment, we have chosen to store as many log tuples together in a sector as can fit. All log tuples are stored

in one or more sectors at the end of a log segment. For simplicity, log tuples never cross sector boundaries. The client data blocks are stored together at the beginning of a log segment. File headers and cluster headers are also stored as client data. However, these have a variable size, but cannot be larger than a disk sector. Therefore, file headers and cluster headers are also packed together in sectors on disk. These headers are also never split across sector boundaries.

We have optimized the disk space usage of a log segment even further. Since the log segment header and trailer of a log segment are much smaller than a sector, it is wasteful to reserve a whole sector for only a few bytes of log segment header or trailer. Therefore, LD fills the first and last sector of a log segment, which contain the header and trailer respectively, with log tuples. This optimization means that log tuples are actually not only at the end of a log segment, but also in the first sector of a log segment, filling the empty space following the log segment header.

5.3 Log Tuple

The previous section has introduced log segments, which mainly contain two types of data: client data and log tuples. In this section, we focus on the log tuples. Every client command that changes the state of the disk is logged when LD executes that command. This logging is done by writing a **log tuple** in the log. The purpose of a log tuple is to describe a client command that has been sent to LD in such a way that it enables LD to redo the command after a crash. Therefore, with the help of the information in the log, LD can recover to a consistent state after a crash. In literature, the logging approach used by LD is referred to as *redo-only* logging (see e.g., [Gray and Reuter, 1993]). We describe the recovery process in detail in Chapter 7.

There are six types of log tuples. Every update operation is described by a log tuple of one of these log tuple types. The log tuple types are listed in Table 5.3. The table also lists the names of the functions in LD's external interface that generate a particular log tuple. Note that sometimes the same log tuple type can be used to describe different types of client commands. The create log tuple is used for the functions that create a disk file or disk cluster. The write log tuple type is used for all functions that change the state of the disk by writing client data. The delete type represents all functions that delete logical blocks, disk files, or clusters. The remaining three types (startaru, commitaru, and abortaru) are used to mark the begin, commit, or abort of an ARU. These six types are sufficient to log all update client commands.

The layout of each log tuple type is based on a generic log tuple, which is shown in Table 5.4. The size of a log tuple is fixed, making it is easier to store and manipulate multiple log tuples in one disk sector. Note that a log tuple does not store client data blocks itself, but only a pointer to them in the field phys_addr. This pointer field is only used when a `ld_write_data`, `ld_set_fh`, or `ld_set_ch` client command is logged. The cluster, file, and offset fields together form a logical block address and, in general, indicate the client data blocks on which an operation is done.

Not every field in this log tuple is used by all log tuple types, and the interpretation of the contents of some of the fields depends on the type of log tuple. However, every log

Table 5.3: Descriptions of the different log tuple types.

Type	Description	Function calls
create	a create operation	`ld_create_file` `ld_create_cluster`

write	a write operation	`ld_write_data`
		`ld_set_fh`
		`ld_set_ch`

delete	a delete operation	`ld_delete_blocks`
		`ld_delete_file`
		`ld_delete_cluster`

startaru	start aru operation	`ld_start_aru`
commitaru	commit aru operation	`ld_commit_aru`
abortaru	abort aru operation	`ld_abort_aru`

tuple type uses the type field to indicate the type of the log tuple. Below, we discuss the other fields in the log tuple when we describe each specific type of log tuple in turn.

Table 5.4: Contents of a generic log tuple.

Field	Description		Size
type	type of log tuple		2 bytes
aru	aru_id		4 bytes
cluster	cluster_id	logical block address	4 bytes
file	diskfile_id		4 bytes
offset	offset		4 bytes
phys_addr	physical block address		4 bytes
count	amount of data written		4 bytes
hdr_offset	start position of header data		2 bytes
	Total size		28 bytes

The create Log Tuple

The first log tuple type we look at is the create log tuple type. Only a few fields of the generic log tuple are used. The relevant fields of a create log tuple are shown in Table 5.5. This log tuple describes the two client commands that create a new disk file or disk cluster, respectively. The type field in the log tuple indicates the type of this log tuple, in this case a create log tuple. The other fields in this log tuple indicate which disk file or disk cluster is created. The cluster and file fields form a disk file identifier or a cluster identifier. Only

if the log tuple is used to log a command that creates a cluster, the file field contains 0. The aru field contains the aru_id of the ARU if the create command is executed within a composite ARU, or 0 if it is executed as a simple ARU.

Table 5.5: Contents of a create log tuple.

Field	Description
type	type of log tuple: create
aru	aru_id
cluster	cluster_id
file	diskfile_id

The write Log Tuple

The fields in a write log tuple are shown in Table 5.6. A write log tuple describes all client commands that write data (logical blocks or header data). The type field in this log tuple indicates that this tuple is a write log tuple. The other fields in this log tuple specify which data blocks are written. The aru, cluster, file, and offset fields form an internal logical block address. The aru field contains the aru_id of the ARU if the command is executed within a composite ARU, or 0 if it is executed as a simple ARU. The other fields of this logical block address indicate the logical block, disk file, or disk cluster that is being operated on.

Table 5.6: Contents of a write log tuple.

Field	Description
type	type of log tuple: write
aru	aru_id
cluster	cluster_id
file	diskfile_id
offset	offset
phys_addr	physical block address of corresponding client block(s)
count	number of logical blocks written or size of header data in bytes
hdr_offset	start position of header data within physical block

A write log tuple either describes a command that writes one or more logical blocks (`ld_write_data`) or a command that writes header data (`ld_set_fh` or `ld_set_ch`). LD can tell the difference by looking at the offset field in the log tuple. In the former case, the offset field contains a nonzero, positive offset. In the latter case, the field contains 0.

If this log tuple describes a `ld_write_data` command, that is, if the offset is positive, the logical block address fields in this log tuple indicate the start of the logical block range that is being written. The count field indicates how many client data blocks are written with this command. The field phys_addr contains the address of the first physical block where these blocks are consecutively stored on disk. These blocks are either written in the same log segment as this log tuple or they were already written in the storage area as part of a *direct segment* before the log tuple was flushed to disk (see Section 5.7). The last field hdr_offset is unused for this command.

If the log tuple describes a `ld_set_fh` or `ld_set_ch` command, that is, if the offset is zero, the rest of the logical block address indicates the disk cluster or disk file for which a header is written. If a file header is written, the file field is positive. If a cluster header is written, the file field is 0. The count field holds the size of the header data in bytes. The combination of the fields phys_addr and hdr_offset hold the exact position on disk where the header data can be found. Recall that, because header data is often small, LD packs multiple file and cluster headers in one physical block during logging. Otherwise, LD would use one block (a 512 byte sector) for each header in the log, which potentially wastes a lot of space, which, in turn, reduces the utilized bandwidth of the disk when writing the log segment to disk. Therefore, the phys_addr field holds the address of the physical block where the header data is located within the log area. The hdr_offset field is the offset where the header data starts within the physical block pointed to by phys_addr. Recall from the last example on page 80 in Chapter 4 that header data are normally located in the metadata area, where they are stored together with LD's metadata. The headers are only written in the log for recovery purposes. Header data are, therefore, not copied out of the log into the storage area by a cleaner process. How header data is written into the metadata area as part of the Mapping is discussed in Chapter 6.

As mentioned before, the header contains two parts: a fixed-size, private part, which is used by LD internally, and a variable-size, public part, which contains the actual client header data. When a client calls `ld_set_fh` or `ld_set_ch` to store data in the public part of a header, the current version of LD always writes both parts of the header in the log. Therefore, the count field is the combined size of the private and public part. In theory, this may lead to extra disk I/O because LD may first have to read the private part of the header from disk again. However, in practice, this private part is already in main memory, because this private part is used to check if the cluster or file exists, which LD must check first before it can store client header data. The advantage of writing both the private and public part is that the entire header information is in the log, which simplifies recovery.

The delete Log Tuple

The delete log tuple type describes the commands that delete a disk cluster, a disk file, or a consecutive range of logical blocks within a disk file. The fields of such a log tuple are shown in Table 5.7. Note that to delete the public part of a disk file header or disk cluster header, the client uses the command `ld_set_fh` or `ld_set_ch` to write an empty public header, and therefore, such a command is described by a write log tuple.

A delete log tuple can be used to describe the three delete commands in LD's interface. If a delete log tuple is used to describe the command `ld_delete_blocks`, which deletes

a range of logical blocks, the cluster, file, and offset fields indicate the starting address of the range of logical blocks that is being deleted. The aru field indicates whether the call is executed within a composite ARU. The last field count indicates the number of blocks that are being deleted.

Table 5.7: Contents of a delete log tuple.

Field	Description
type	type of log tuple: delete
aru	aru_id
cluster	cluster_id
file	diskfile_id
offset	offset
count	number of logical blocks deleted

If a delete log tuple is used to describe the command `ld_delete_cluster` or the command `ld_delete_file`, which delete a whole disk cluster or disk file, respectively, then the logical block address in the log tuple specifies which disk file or cluster is being deleted. In the first case, the offset field is 0, and in the latter both the offset and the file fields are 0. The count field is unused for these two commands.

The ARU Log Tuples

The remaining three types of log tuples (startaru, commitaru, and abortaru) are used to mark the begin, commit, or abort of an ARU. These tuples all have the same structure, which is shown in Table 5.8. The startaru, commitaru, and abortaru log tuples use only the aru field, which holds the aru_id of the ARU that is started, committed or aborted, respectively. During the replay of the log after a crash, LD can determine whether an ARU completed (committed or aborted) or if the crash had occurred before the ARU could complete, in which case only a startaru is present in the log without a corresponding commitaru or abortaru.

Table 5.8: Contents of the three ARU log tuples.

Field	Description
type	type of log tuple: startaru, commitaru or abortaru
aru	aru_id of started, committed or aborted ARU

5.4 Accumulating Data in the In-Core Segment

LD writes log segments in one large disk operation to optimize the disk bandwidth utilization. To this end, client data blocks are first accumulated in main memory in the **in-core segment**. Conceptually, the in-core segment is an array of main-memory data buffers which are used to hold client data as well as log tuples. In practice, it is implemented as an array of pointers to buffers in LD's cache. Currently, the maximum amount of memory reserved for the in-core segment is 1 MB.

Every client command that updates data on disk results in a corresponding log tuple being appended to the list of log tuples in the in-core segment. Any client data blocks that are written as part of the execution of the command are also put in data buffers in the in-core segment. The blocks in the in-core segment are usually written as log segments in log area slots in the log area. The exception to this is when LD writes the blocks as direct segments to disk instead of log segments (see Section 5.7). Since the slots in the log area are allocated in a round robin fashion, the location on disk where the next log segment will be written is already known while the in-core segment is being filled with client data blocks and log tuples. As a consequence, LD can already determine the physical disk address of each of the client data blocks in the in-core segment while it is being filled. These addresses can then be used in the generation of the corresponding log tuples. In the following sections, we will see that LD can later change the order of the client data blocks in the in-core segment to influence the physical layout of those blocks in a log segment on disk. When LD does so, it must also adjust the physical addresses in the log tuples to represent the new ordering in the in-core segment.

To illustrate how write commands are stored and logged in the in-core segment, let us look at the following example. We simplify the example by looking at a system with only one command stream. Figure 5.2(a) shows four write commands that are sent to LD as simple ARUs. Suppose that they are sent by two clients. Each client writes 256 KB of data (i.e., 512 logical blocks in LD, which are one sector each) to an existing disk file; one client writes to disk file (8, 1) and the other to disk file (8, 2). Both clients write their data in two separate commands of 128 KB (i.e., 256 logical blocks) each. In the figure, we have hatched the commands of the two clients differently, so we can distinguish between the data from both clients.

Suppose that these commands arrive at LD interleaved, as shown in the figure. Because the commands arrive in the same stream, LD handles these requests in the order they arrive, as was discussed in our example in Section 4.2. The client data blocks are put in the in-core segment, which is shown in Figure 5.2(b). We have assumed that the in-core segment already contains data, so the data of our four commands is drawn somewhere in the middle of the segment.

For clarity, we have not included the blocks that hold the log tuples in the in-core segment, but show the log tuples of the in-core segment separately in Figure 5.2(c). Four new tuples have been added for the four write commands. In this case, each command writes 256 blocks of 512 bytes each, which is 128 KB. We only show the most relevant information of the contents of each log tuple. The first letter indicates the type of the log tuple. Here, we have used the letter *W* to indicate a write log tuple. This is followed by the logical block address, in which the aru_id is 0 because the commands are sent outside

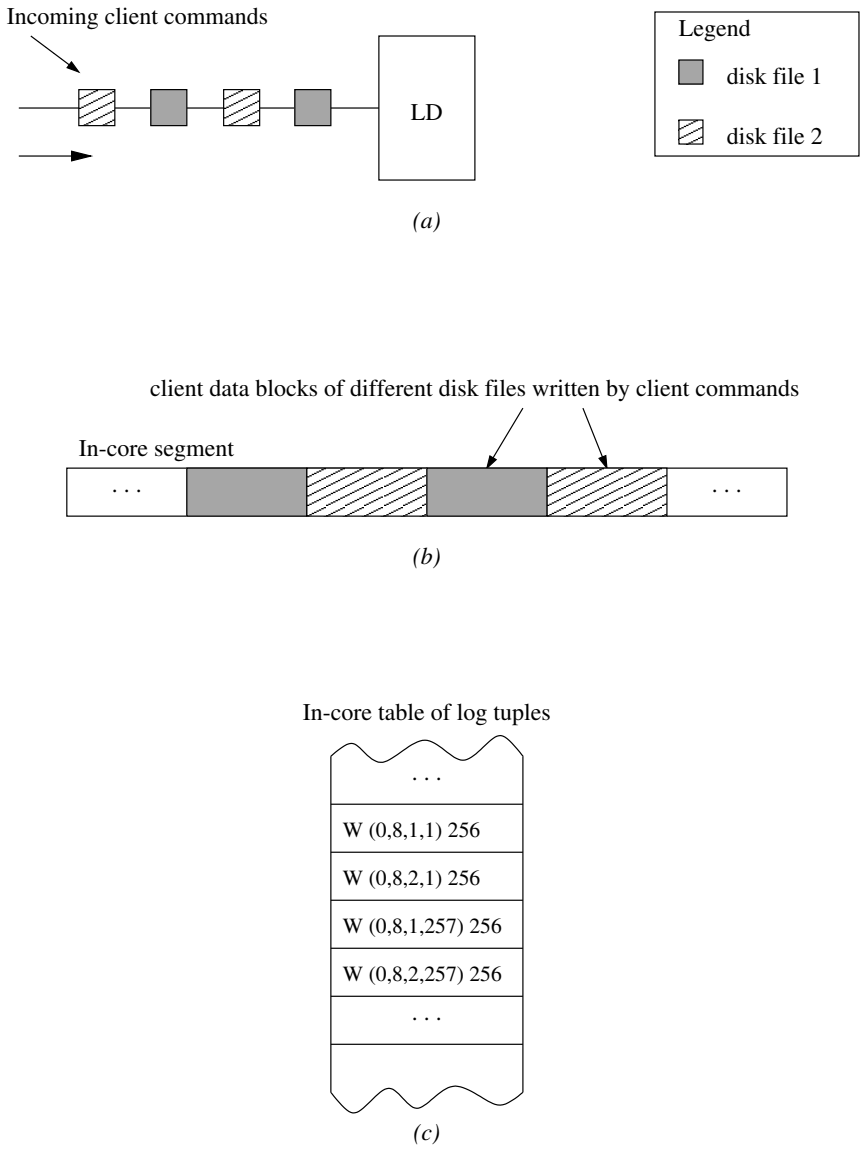


Figure 5.2: Accumulating data in an in-core segment. (a) Four write commands are sent to LD. (b) Data blocks of the write commands are put in the in-core segment. (c) For each command a corresponding log tuple is generated.

an ARU, and by the amount of blocks written. We have left out the phys_addr field which refers to the client data blocks that have been written.

Commands whose data and log tuples reside in the in-core segment have been executed, but are not recoverable yet because they have not been written to disk yet. The advantage of first storing data in main memory is that the data in the in-core segment can be easily manipulated. LD can manipulate and sort the data in the segment such that it can write the segment to disk more efficiently. There are at least three optimizations LD can do with the data while it is still in the in-core segment. The first optimization deals with clustering blocks in the in-core segment. The second one deals with data blocks that are overwritten in the in-core segment, and the last optimization concerns writing direct segments. These three optimizations are the subjects of the following three sections.

5.5 Clustering Data in the In-Core Segment

The first optimization is that LD can adjust the order of the data blocks in the in-core segment before it writes them to disk as a log segment. The goal of ordering the data blocks is to physically cluster blocks that belong together in the log segment. This makes future sequential access to these clustered blocks more efficient. There are two reasons to do this.

First, unlike more traditional logs, LD's log is an integral part of the storage space. Therefore, even though the locations of the blocks in the log are only temporary until a cleaner has time to move them to the storage area, these blocks are accessible for normal read commands from clients. As a result, physically clustering the blocks that are likely to be accessed sequentially is good for LD's read performance.

The second reason is to speed up the cleaning step which moves the data blocks from the log area to the storage area. If the blocks that must be stored well-clustered in the storage area are already clustered in the log, a cleaner process can copy the blocks out of the log more efficiently than if they are spread over the log segment.

For completeness, we add that this is probably only a minor optimization. In practice, we expect that most data blocks that have been written in the log and have not yet been cleaned into the storage area will still reside in the cache of LD. Therefore, accesses to such blocks, either resulting from a client read command or LD's internal cleaner process, are likely to be satisfied from the cache, so actual disk I/O is not necessary.

5.6 Overwriting Data in the In-Core Segment

The second optimization LD implements is the removal of obsolete data blocks and log tuples from the in-core segment, which avoids the overhead of writing stale data to disk as part of the log. For example, data blocks in the in-core segment become obsolete when they are overwritten. Let us consider the following example. Suppose that a client issues two commands to write the same block. Let us also assume that LD executes the second write command while the block that was written by the first write command is still in the in-core segment. Conceptually, the second write command overwrites the logical block of the disk file that was written by the first write command. Therefore, LD can also

overwrite the corresponding data block in the in-core segment to avoid useless work. The block written by the first write command as well as its log tuple are removed from the in-core segment.

The advantage of this removal is that the in-core segment only contains valid data. However, notice that this also means the log tuples in the log actually do not hold a historically complete representation of all executed commands, but only hold a shortened, but equivalent version of the command execution history. In other words, the final result of replaying the log tuples of the shortened history is the same as the one that results from replaying the complete history. We will come back to this correctness issue in Section 5.6.2.

Example 1: Overwriting a Single Block

The situation of overwriting a single block in the in-core segment is not uncommon. For example, consider a file system on top of LD that uses a disk file to implement a directory in a file system. In other words, that disk file holds the information that tells which files reside in the corresponding directory. Let us call this disk file the *directory disk file*. This disk file can be seen as a table of entries. Each entry consists of a pair holding a *file name* and an *i-node number*. Every time a new file-system file is added or removed from the directory in this file system, the contents of the corresponding directory disk file change, which means that its data blocks are updated. Since an entry in the directory disk file is small, a single block of that disk file can hold many entries. Therefore, a human user who creates many new file-system files in the same directory causes many block overwrites to a single block of the directory disk file. For example, this happens when a human user gives a command to a file system to copy a file-system directory containing many files. Without the optimization that allows LD to overwrite data in the log, each creation of a new file in the directory would append another version of the same logical block of the directory disk file to LD's log. As a consequence, the in-core segment would fill up quickly and LD would write many obsolete directory blocks in its log, which results in an inefficient way of using the available disk bandwidth.

Example 2: Overwriting Multiple Blocks

To further illustrate the effect of overwriting data in the in-core segment, let us look at a more complex example. Suppose that a client sends three commands to LD. The first command that is sent to LD writes the first 64 blocks in an empty disk file (8,1). Second, a delete command to delete 8 blocks at positions 17 through 24 of that disk file is sent to LD. Last, another write command is sent that overwrites 24 blocks at positions 33 through 56 in the same disk file. Assume that during the execution of these commands the in-core segment is not yet written to disk, so all commands are still only executed in main memory. The first write command results in 64 data blocks in the in-core segment and a write log tuple. The log tuple is shown in Figure 5.3(a). After executing the second command, which is a delete, LD can remove 8 data blocks from the in-core segment. However, we must now change the first write log tuple, that stated that 64 blocks were written. This log tuple is split in two new log tuples. One is a write tuple for 16 blocks, the other for the remaining 40 blocks at position 25, as is shown in shown in Figure 5.3(b). The delete

tuple is also generated because during recovery LD must remember to remove blocks 17 through 24, which may have existed before they were overwritten in the first write command. We use the letter *D* to denote a delete log tuple. Note that in this particular example, we were, theoretically, not obliged to generate this tuple, because in our example the disk file was empty at the beginning, however, in general, this may not be the case.

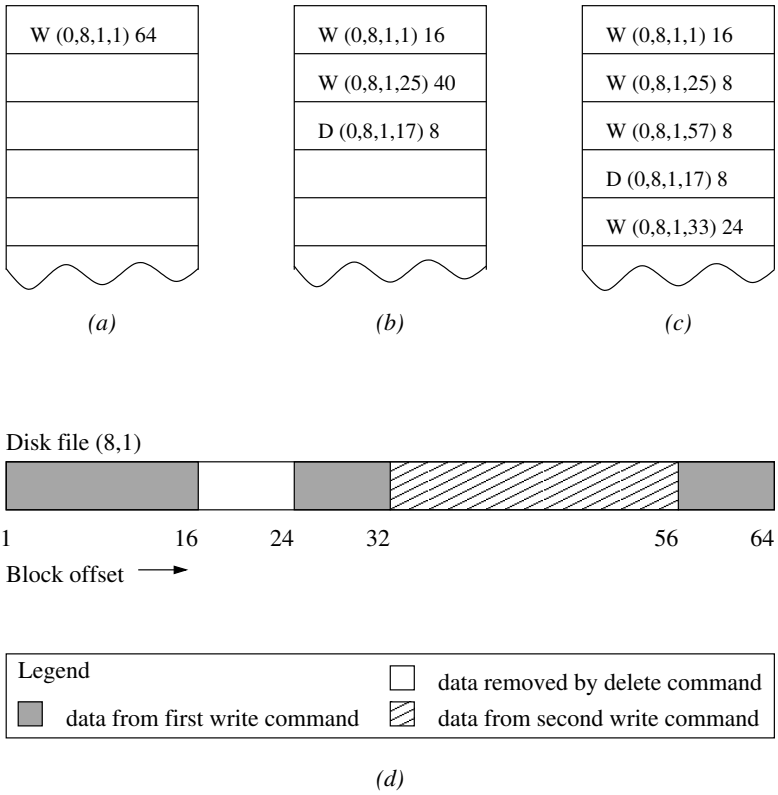


Figure 5.3: Overwriting data in the in-core segment. (a) Write 64 blocks to disk file (8,1). (b) Delete blocks 17 through 24 from the disk file. (c) Overwrite blocks 33 through 56 in the disk file. (d) The end result of the disk file.

The last command overwrites 24 blocks. These 24 blocks overwrite the existing data blocks of the first write command in the in-core segment. This time, the existing log tuple is split again and a new log tuple for the 24 blocks is appended to the list of log tuples (Figure 5.3(c)). LD can store the 24 blocks in the in-core segment by overwriting the contents of the original 24 blocks and reuse their buffer space. Figure 5.3(d) shows the resulting disk file. The blocks that have been written as part of the same write command have been shaded or hatched likewise.

Example 3: Temporary Files

Another situation where LD can easily avoid writing obsolete data to the log segment is the following. Suppose a client sends a combination of commands in which it creates a disk file, fills it with data, reads that data and deletes the same disk file again, in succession. Let us assume the diskfile_id of that new disk file was not in use just before the disk file was created, and all commands are executed within one in-core segment. For instance, applications that use temporary files exhibit such behavior. In this case, LD can completely remove any reference to the disk file and its blocks from the in-core segment after LD has executed the delete command for the disk file.

For completeness, note that LD cannot remove all log tuples from the in-core segment if not all commands are executed within the same in-core segment. If the create log tuple had already been written to disk in a previous log segment, then LD must also leave the delete log tuple in the in-core segment and eventually write it to disk in order to cancel the previously written creation log tuple.

Also notice that if the disk file already existed (its create tuple was written in a previous log segment) and was deleted just before the client reused the same diskfile_id for a temporary file, LD must also leave the last delete log tuple in the in-core segment. LD cannot remove this log tuple because, in this case, another delete log tuple would have been in the in-core segment just before the create log tuple. If LD removed all these log tuples from the in-core segment, including both delete log tuples, LD would recover to an incorrect state in which the disk file still exists.

5.6.1 Optimizing Combinations of Commands

The above mentioned examples are combinations of commands that act on the same data blocks. One command writes data into one or more data blocks, and a subsequent command overwrites the same data blocks (or part of those blocks), or deletes them. In each of these situations, LD can deal with them efficiently by overwriting or deleting the affected data while it is still in the in-core segment. There are a number of command combinations that LD can deal with in a similar fashion. These combinations are summarized in Table 5.9. The first column of the table lists commands, whose data in the in-core segment can be (partly) overwritten or even removed if, at a later point in time, it is followed by a command in the second column. Note that this only applies if both commands act on the same or overlapping data ranges. The exact action that LD takes in each case depends on the particular combination of commands and how much they overlap. For example, in our example of Figure 5.3, data was overwritten or deleted, which resulted in adjusting log tuples and removing the corresponding data blocks from the in-core segment. However, our last example, which created and deleted a disk file within the same in-core segment, removed every trace of the commands from the in-core segment.

The effect of overwriting data blocks in the in-core segment can also result from a commit of a composite ARU. Commands that are sent within a composite ARU create uncommitted versions of data blocks, which are different from the committed versions. As a result, both a committed and an uncommitted version of the same block can be present in the in-core segment at the same time. The one does not overwrite the other. However, a commit of such an ARU, causes the uncommitted version of that block to

Table 5.9: Combinations of commands that overwrite data in the in-core segment.

First command	Second command	Description
ld_write_data ld_delete_blocks	ld_write_data	Writing blocks that have just been written or deleted.
ld_write_data ld_delete_blocks	ld_delete_blocks	Deleting blocks that have just been written.
ld_create_file ld_write_data ld_delete_blocks ld_set_fh ld_delete_file	ld_delete_file	Deleting a disk file allows LD to remove all other commands that access that disk file from the in-core segment.
ld_create_cluster ld_create_file ld_write_data ld_delete_blocks ld_set_fh ld_set_ch ld_delete_cluster	ld_delete_cluster	Deleting a disk cluster allows LD to remove all other commands that access that cluster or any disk file within that cluster from the in-core segment.
ld_set_fh	ld_set_fh	Overwriting a file header.
ld_set_ch	ld_set_ch	Overwriting a cluster header.

overwrite the committed version. In other words, a commit of a composite ARU can also result in data blocks that are overwritten in the in-core segment if that segment also contained the previously committed version of one or more blocks being written in that ARU.

To clarify this, let us look at the following example. Suppose a client writes the first block of disk file 5 using a simple ARU. Subsequently, another client overwrites that same block within an already running composite ARU with aru_id 8. Both blocks have different internal logical addresses ((0,1,5,1) and (8,1,5,1)), and therefore, both blocks are in the in-core segment, each with their own log tuple. These log tuples are shown in Figure 5.4(a). However, when the ARU is committed, the uncommitted block with address (8,1,5,1) atomically becomes the new committed block, overwriting the previously committed version in the in-core segment. In this case, a commitaru log tuple is added to the in-core segment to signify that ARU 8 has committed, and the old committed block with address (0,1,5,1) and its log tuple can be removed from the in-core segment. The letter *C* is used to denote the commitaru log tuple. This situation is depicted in Figure 5.4(b).

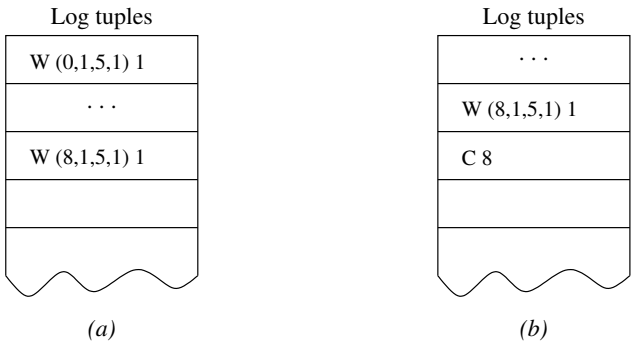


Figure 5.4: Overwriting data in the in-core segment with a composite aru. (a) Block (1,5,1) is written twice, first outside an ARU, and then within ARU 8. (b) After ARU 8 commits, the uncommitted version overwrites the committed version; its log tuple is removed.

Note that the table does not list all possible combinations of overlapping commands because cancelling commands in the in-core segment is not always possible. For example, a deletion of a disk file followed by a creation of the same disk file is not listed in the table because the effects of the delete are not completely cancelled by the create command. To illustrate this, consider the following situation. Suppose that a client first issues a command to delete an already existing disk file, and then immediately creates a new disk file with the same diskfile_id. The delete command is represented by a delete log tuple in the in-core segment. This command causes all log tuples that are currently in the in-core segment and refer to the same disk file to be discarded. The subsequent create command is represented by a create log tuple. However, these two log tuples do not cancel each other, because the delete command also deletes all existing blocks in that disk file. In short, the delete and create log tuples cannot be left out of the log. Otherwise, during recovery, LD would not know that the disk file has been deleted and subsequently created again. As an

optimization, we could change the semantics of a create log tuple so that it also implicitly deletes any existing blocks within the disk file or cluster before creating it. This way LD could leave out the delete, but leave the create log tuple. However, the gain is not worth the trouble, and therefore, we have not chosen this design.

5.6.2 Changing Log Tuples in the In-Core Segment

In the three examples mentioned in the beginning of Section 5.6, we see that LD changes log tuples in the in-core segment. These changes include splitting existing log tuples and removing old log tuples. As a result, not all log tuples originally added to the in-core segment end up being written to disk. It is clear that manipulating log tuples may have an effect on the state that is recovered after a crash since during recovery LD constructs the recovered state by replaying log tuples. Therefore, LD must make sure that when it changes log tuples in a log segment, it does not also change the outcome of the recovery process.

As mentioned before, the log tuples in the log are replayed during recovery to restore the contents of the disk to a consistent state. LD writes log segments atomically to disk, which allows LD to recover the log segmentwise. In other words, during recovery *all* log tuples within a log segment are replayed or none of them are. As a consequence, the log tuples in a log segment only have to represent the 'end result' of the commands that were executed within that log segment. Any intermediate state that existed while that log segment was being created as an in-core segment, does not have to be represented in the log tuples.

If we look from the viewpoint of the data blocks, each log tuple represents a change to one or more data blocks on disk. Each operation that is logged in the in-core segment changes one or more data blocks; it either writes new versions of data blocks or it deletes them. When LD writes a log segment, it is only interested in the last change to each data block that has been made within this log segment since log segments are written and recovered atomically. These changes are represented by log tuples. In other words, LD is only interested in log tuples whose effects were still visible at the moment that the in-core segment was written to disk as a log segment. Log tuples whose effects have been superseded by newer log tuples are obsolete and do not have to be included in the log segment.

Note that LD is not obliged to change (partially) obsolete log tuples. It is perfectly correct for LD to leave obsolete log tuples in the in-core segment unchanged and to eventually write them to disk. When LD needs to replay log tuples during recovery, LD replays all log tuples, including these (partially) obsolete log tuples, which will result in the correct state in which all client updates have been incorporated. The difference between this log and an optimized log in which these (partially) obsolete log tuples have been changed, is that with the unoptimized log LD will do some useless work during recovery; the (partially) obsolete log tuples would have lead LD to do some work that would have been undone by a subsequent log tuple.

Rules for Changing Log Tuples

How does LD manage the log tuples and guarantee that the outcome of replaying the log tuples is correct? The answer is that LD abides by the following four rules:

Rule 1: All log tuples in the in-core segment are kept in the order of arrival. New log tuples are appended only at the end.

Rule 2: When appending a new log tuple, earlier log tuples that refer to one or more of the same data blocks as the new log tuple, may be altered or removed when possible.

Rule 3: All log tuples in the in-core segment are always atomically written to disk in a log segment.

Rule 4: The data blocks that a log tuple refers to must always be written to disk before or at the same time as that log tuple is written to disk.

The first rule states that, in principle, the order of log tuples is respected. This way, during recovery LD can replay the log tuples in the same order that they arrived, which will result in a state that incorporate the changes described by the log tuples in the correct order.

The second rule states that LD can change old log tuples whenever a new log tuple makes (parts of) these old log tuples obsolete. However, when LD changes log tuples, it may not change the relative order of the log tuples because that is the order in which they will be replayed during recovery. The end result of replaying the log tuples must be the same as if LD had not changed log tuples.

When LD is allowed to change log tuples, it is important that LD redoes all log tuples in the log segment during recovery, otherwise the end result may not be consistent. This is why the third rule states that all log tuples of the in-core segment must reach the disk atomically. Since LD writes log segments atomically to disk, there is no danger that only a part of the log tuples that were in the in-core segment have reached the disk, and LD can always recover all log tuples.

The last rule is a general rule. It states that the data blocks to which each log tuple refers, must also be on disk when that log tuple reaches the disk. Otherwise, a crash after writing the log tuple but before writing the corresponding data blocks would leave LD in a position where it cannot redo the command because part of the information is missing. Therefore, either the data blocks must have been written to disk before, or it reaches the disk as part of the same atomic log segment write as the log tuple. This requirement is important when LD writes direct segments, which contain client data blocks and are written to disk outside the log.

How does LD implement these rules? We start by explaining how LD implements the third rule, which was actually already explained in a previous section. Namely, LD writes log segments atomically to disk, and therefore, the log tuples inside them are also written atomically to disk. The fourth rule is implemented by making sure that all dirty client data blocks from LD's internal buffer cache that are not part of the log segment, but are referred to by log tuples within that log segment, are flushed to disk before writing the log segment with log tuples. Examples of such dirty client data blocks are the blocks of a direct segment.

The first and second rule state that LD is allowed to change log tuples when (parts of) these log tuples are made obsolete by a new log tuple, as long as it leaves the order in which log tuples arrive intact. Maintaining this order is relatively simple. Each time an updating client command is executed, a corresponding log tuple, describing the command and the affected client data blocks, is generated and appended to the list of log tuples in the in-core segment. Subsequently, LD checks whether one or more previously generated log tuples in the in-core segment refer to any of the data blocks, that have just been updated. If so, then these older log tuples are completely or partially obsolete since the changes in these older log tuples have been superseded by the current log tuple. LD may decide to change these older log tuples so that the block ranges they refer to do not include these currently updated data blocks anymore. There are two actions LD can do to such a (partially) obsolete old log tuple: remove it, or replace it with one or more new log tuples. Whenever the log tuple is made completely obsolete by a new log tuple, it can be removed. If the log tuple is only made partially obsolete, it can be replaced by one or more log tuples that only describe those parts that are still valid.

As stated earlier, when LD changes log tuples, LD may not change the relative order of the log tuples, otherwise LD would violate the first rule. LD can remove an obsolete log tuple with little problems because this does not change the relative order of the remaining log tuples. This happens when the older log tuple does not refer to any valid data blocks anymore because all the blocks it used to refer to have been deleted or rewritten by other commands. An example of this is when a client updates the same block twice. The log tuple describing the first write command is obsolete because it is completely superseded by the second write, and can thus be removed.

When LD replaces a log tuple with one or more log tuples, LD must update the log tuple in-place. In other words, the new log tuples take the position of the log tuple that is being replaced. The relative order between these new log tuples must be chosen such that executing them in that order gives the required result. LD chooses the new log tuples such that they refer to disjoint block ranges. That way, replaying these log tuple does not (over)write or delete each others data blocks. As a result, the order in which these new log tuples are executed is irrelevant. An example of this was Example 2 in the beginning of this section on page 95, which was illustrated by Figure 5.3. In that example, we showed the log tuple, that was generated by writing the first 64 blocks of an empty disk file. Subsequently, we deleted a consecutive range of 8 blocks from that file. This delete command generated its own log tuple, but also made the first log tuple partially obsolete. LD replaced this log tuple with two new log tuples, such that they do not refer to the 8 deleted blocks anymore.

However, when LD decides to change log tuples, it must be careful to leave the remaining log tuples consistent with the changes it has made. For instance, in Example 3 on page 97, we discuss how LD can efficiently deal with temporary files. If within one in-core segment a disk file is created, filled with blocks, and deleted again, LD may decide to remove all corresponding log tuples. However, it cannot remove only the create log tuple and leave the other log tuples because during recovery LD would get confused when it sees the remaining log tuples that write data blocks into a disk file for which LD has not seen the corresponding create log tuple. LD should remove all log tuples concerning that disk file when possible.

5.6.3 Log Tuples and Multiple Streams

Until now we have only considered commands that have been issued within one stream. Luckily, the situation with multiple streams is not very different. Notice that a log tuple does not contain a field to record the stream_id in which the corresponding command was sent. The reason for not storing the stream_id in a log tuple is that this information is not necessary during recovery. When clients send commands over multiple streams, LD must respect the order of executing the commands per stream. At run time, LD determines the order in which the commands are executed. It is this execution order that determines the state to which must be recovered after a crash. Notice that the log tuples of those commands are also generated and put in the in-core segment in that order. Even though LD can manipulate these log tuples as part of the optimization process, this does *not* alter the end result of replaying the log tuples after a crash. In other words, the execution order of the original client commands is also incorporated in the log tuples of the log. Therefore, replaying the log tuples in the log during recovery will bring the disk back into a state in which the results of client commands are incorporated in the order they were executed. As a result, it is not necessary to store the stream_id in a log tuple because during recovery it is not needed to determine the execution order anymore; the commands have already been serialized at run time.

5.7 Forming Direct Segments

Data blocks that are written in a log segment in the log area on disk must at a later point in time be moved out of the log area into the storage area to achieve the desired clustering of the data blocks. This reorganization means that these data blocks are actually written twice. The reason for this two step write process is to provide good integrity guarantees (i.e., no in-place updates) as well as to combine good write performance (i.e., collective writes) with good read performance (i.e., clustering data on disk).

The third type of optimization that LD can do to data in the in-core segment before it is written to disk consists of avoiding the two step write process and still accomplish the same three goals with only a single write operation, which saves the relatively costly reorganization afterward. The main idea is that LD writes data blocks that have accumulated in the in-core segment directly in the storage area, bypassing the log. Unfortunately, this is only possible in certain cases. We call writing a number of blocks directly from the in-core segment into the storage area, bypassing the log, writing a **direct segment**.

By-passing the log is only worthwhile if the client data blocks written outside the log satisfy two conditions:

Condition 1: The number of client data blocks is large.

Condition 2: The client data blocks do not need reorganizations soon after they are written into the storage area.

If the number of blocks is not large enough, the seek and rotational delay for a separate write operation in the storage area would lower the utilized disk bandwidth too much. If reorganization is necessary soon afterward, then the benefit of bypassing the log is lost.

There are two reasons that make reorganizations of data blocks on disk necessary:

(1) To improve the clustering of client data blocks.

(2) To create contiguous free space by defragmenting the disk.

The first reason has been mentioned many times before: data blocks that are written in the log are copied out of the log into the storage area by a cleaner process in order to maintain the clustering of those blocks with other blocks of the same disk file. However, this is not the only reason that the cleaner copies data blocks out of the log. The other reason is to clean the log in order to create free space in the log area, so that new log segments can be written. This brings us to the second reason for reorganization: creation of contiguous free space by defragmenting the disk. Cleaning the log area is one example of creating contiguous free space on disk. We will shortly see that this is also necessary in the storage area where direct segments are written.

The question we will answer in this section is what type of client data blocks satisfy the conditions to be written into the storage area directly? To answer this question, we distinguish between four types of data blocks in the in-core segment. We look at the client data blocks of each disk file separately and look at two criteria:

- **Clustered/Unclustered** — Is physical clustering required for this disk file?
- **Large/Small** — Is there a large, possibly consecutive, number of data blocks for this disk file present in the in-core segment?

Table 5.10: When to write client data blocks of a single disk file into a direct segment or a log segment.

	Clustered disk file	**Unclustered disk file**
Large number of data blocks	Direct segment	Direct segment
Small number of data blocks	Log segment	Log segment

These criteria lead to four different categories into which the blocks of each disk file can be put. For each disk file, LD determines the type of that disk file's blocks in the in-core segment, and decides based on that whether to write those blocks directly into the storage area as a direct segment or into the log as a log segment. Table 5.10 shows the four categories, and how LD writes data blocks of these categories to disk. In the following subsections, we will look closer at each of the four categories, and show whether they satisfy the conditions to write data blocks to disk as a direct segment or not.

5.7.1 Large Range of Clustered Blocks

The first category consists of client data blocks of a single disk file for which clustering is requested, and furthermore, a large consecutive number of these blocks is to be written. In

this subsection, we examine whether it is worthwhile to write client data blocks that form a large clustered block range directly into the storage area via a direct segment instead of in the log via a log segment. As stated above, LD should write these data blocks within a direct segment only if it can find enough data blocks to fill a direct segment, and if reorganizations are likely to be unnecessary after these blocks have been written to disk as a direct segment.

The first condition is satisfied by definition. Data blocks that form a large consecutive range of single disk file are large enough to fill a direct segment. Therefore, writing those blocks as a direct segment will utilize the available disk bandwidth sufficiently. We will come back to the issue how much data is actually necessary to form a direct segment in Section 5.8.

Next, we must check whether the second condition is satisfied as well. This condition states that after writing the client data blocks as a direct segment into the storage area, no reorganizations should be necessary soon afterward. Recall that there are two reasons to reorganize data blocks: the restoration of the clustering and the creation of contiguous free space. Since, in this case, the client data blocks form a large consecutive range of a single disk file, and therefore, are already sufficiently clustered, there is no need to reorganize them in order to restore clustering after they have been written to disk. If we write the blocks of a large disk file via multiple direct segments, the disk file will be stored segmentwise. Although the segments themselves may be stored relatively far apart on disk, the sequential read performance when reading this file will still be acceptable because the data is clustered in segment-size chunks.

In general, the need for contiguous free space (the second reason for reorganization) in the near future is impossible to predict accurately. When direct segments are concerned, preventing fragmentation is important because otherwise LD cannot find free spaces large enough to hold new direct segments. However, notice that in this case, the direct segment is filled with consecutive client data blocks of a single disk file. It is unlikely that over time such a direct segment will suffer from much internal fragmentation; files are usually stable, or are deleted or rewritten as a whole, and therefore, the risk of individual blocks being deleted is small.

As a result, it is likely that LD can manage direct segments filled with client data blocks that form large clustered block ranges as large units, and not as individual blocks. Consequently, LD can manage the disk space occupied by such direct segments relatively simply; little defragmentation is necessary to find new space for new direct segments.

Therefore, we can conclude that it is worthwhile to write a large range of consecutive client data blocks of a single disk file in a direct segment to disk. In other words, if LD finds such a range of blocks in the in-core segment, LD does not have to write these blocks into the log segment first in order to achieve good write performance. LD can simply write those consecutive data blocks in one large write operation directly into the storage area, bypassing the log, and thereby, avoid having to write them twice. However, to prevent overwriting any of those data blocks in the storage area (i.e., an in-place update), LD must find a contiguous range of free space in the storage area large enough to hold those blocks.

5.7.2 Small Range of Clustered Blocks

Data blocks belonging to this category are from a single disk file for which physical clustering is requested. However, there are not enough consecutive data blocks available in the in-core segment to fill a separate direct segment. This already means that blocks of this category do not fulfill the first condition: there are too few blocks to warrant a direct segment. We can simply remedy this by taking together the blocks, that also belong to this same category, but belong to other disk files. This may provide LD with enough blocks to fill a direct segment, but, unfortunately, we still do not fulfill the second condition: no reorganizations of the direct segment.

Clustering for blocks of this category is important because a client has requested physical clustering of this disk file. When we write these blocks to disk as a direct segment, LD cannot consider them sufficiently clustered because each disk file in that direct segment has too few blocks present in that segment. Therefore, LD must reorganize these blocks afterward to restore their clustering properties. For instance, a reorganizer process could for each block in the direct segment determine where the rest of that block's disk file is located in the storage area, and move this block closer to that location.

In conclusion, it is not worthwhile to write blocks of this category directly into the storage area as a direct segment because further reorganizations are necessary. Recall that LD writes log tuples for the commands that wrote these client data blocks, in the log area. Therefore, LD also writes blocks of this category in the log together with the log tuples, which avoids an additional seek.

5.7.3 Large Number of Unclustered Blocks

The third category consists of client data blocks of a single disk file for which clients have *not* requested physical clustering. Since physical clustering is unimportant for such blocks, any place on disk is suitable without a need to reorganize them later on to restore clustering properties. Recall from Section 3.3 on page 50 that the request to physically cluster disk files is under control of LD's clients, such as file systems, and not human users. Therefore, even though human users may be inclined to desire physical clustering for all their data, a file system is not as it will try to optimize the overall performance of the system.

The first condition that must be satisfied to warrant a separate disk write operation to write those blocks in a direct segment into the storage area is that enough data blocks must be available. This condition is also fulfilled by definition: there are enough blocks of a single disk file available to fill a direct segment. Note that, unlike the category 'large range of clustered blocks', the blocks of this category do not have to be logically consecutive because physical clustering is not important.

The second condition was that reorganization of the direct segment should be unnecessary. Because we are dealing with unclustered blocks, reorganizing to restore their clustering properties is unnecessary, as stated above. The other reason for reorganization was the creation of contiguous free space. Since the blocks of this category are from a single disk file, the chances of internal fragmentation in a direct segment filled with these blocks is relatively small. We can use the same argument as we did in a previous subsection which discussed the category 'large range of clustered blocks'. Therefore, we can

conclude that blocks of this category also form good candidates to be written directly into the storage area within a direct segment.

5.7.4 Small Number of Unclustered Blocks

Similar to the previous category, this category is formed by blocks of a single disk file for which no client has requested physical clustering. However, unlike the previous category, there are not enough blocks available of a single disk file to fill a direct segment. By accumulating blocks of the same category, but from different disk files, LD can fulfill the first condition and collect enough data blocks.

For example, consider a system where many clients update many small disk files, each of which does not have to be stored clustered. These updates result in data blocks in the in-core segment. These data blocks belong to the category 'small number of unclustered blocks'. LD could accumulate these data blocks and form and write a direct segment. Such a direct segment would contain blocks of many different disk files.

Unfortunately, the risk of internal fragmentation in a direct segment filled with data blocks of multiple disk files is relatively high. Recall that a file is usually stable or is deleted or rewritten as a whole; blocks are seldom deleted individually. Therefore, on the one hand, a direct segment filled with blocks of a single disk file is not likely to suffer much from internal fragmentation. On the other hand, however, a direct segment filled with blocks from multiple disk files has a higher chance of internal fragmentation because a deletion of one of those disk files results in fragmentation in that direct segment.

Therefore, in general, a direct segment with blocks of this category, but from multiple disk files, will likely suffer from internal fragmentation. This fragmentation must be remedied in order to find empty space for writing future direct segments. This requires a reorganizer process to move blocks on disk in order to defragment the disk, which is overhead. Therefore, in general, it is not clear whether writing such blocks via direct segments is more efficient than writing them via the log. This depends on how much the fragmentation of such direct segments and LD's need for free space necessitate reorganizations in the future. Experiments with real applications on top of LD are necessary to determine how best to deal with blocks of the 'small number of unclustered blocks'-type. However, in our current design, we have chosen to write such blocks within log segments to disk.

5.7.5 Other Candidates for Creating Direct Segments

Notice, that we make a rather rough categorization of client data blocks in four types. Based on this categorization, we have come to the conclusion that we write client data blocks of the categories 'large range of clustered blocks' and 'large range of unclustered blocks' as direct segments. Blocks of the other two categories are written within a log segment and need to be copied from the log into the storage area by a cleaner. However, there may be other cases in which we can successfully form and write direct segments, and avoid unnecessary reorganizations. Therefore, it could be worthwhile to make a finer distinction between types of data.

For example, consider a situation where an application writes many small disk files, for which physical clustering is requested. Because complete disk files are written (not

just small parts of them), the in-core segment will be filled with perfectly clustered data blocks of many disk files. Therefore, when LD writes these blocks to disk within a direct segment, LD does not need to reorganize them to restore clustering. However, according to our current categorization, these blocks will be written to disk within the log, as they do not fall under the 'large clustered block ranges' category. Whether it is worthwhile to write them within a direct segment, again, depends on the need for future defragmentation.

Another way to create direct segments is to mix data blocks from different categories. For example, what if LD can find a large range of consecutive blocks of a single disk file in the in-core segment. Unfortunately, it is just a couple of blocks short of forming a direct segment. In this case, it may be worthwhile to fill up the direct segment with some unclustered blocks, or maybe a small, but complete, disk file.

There are many combinations possible to create direct segments. The main question remains whether writing client data blocks in a direct segment is more efficient than writing them in the log. The advantage of the scheme we have chosen (i.e., only a large range of clustered blocks and a large range of unclustered blocks are written as direct segments) is that the disk space that holds direct segments is relatively easy to maintain as fragmentation is not a big problem. We will discuss where LD stores direct segments in Chapter 8.

5.7.6 Rules to Form Direct Segments

In this section, we look at how direct segments are formed. When the buffers of the in-core segment fill up, LD tries to form and write direct segments. In the current version of LD the size of a direct segment is 256 KB, which is large enough to guarantee effective disk bandwidth utilization when transferring data segmentwise, which was discussed in Chapter 2. After writing a direct segment, LD can remove the written data blocks from the in-core segment, which allows LD to accumulate more data in the in-core segment again. However, these blocks are recently written client data blocks, and therefore, in practice, LD usually keeps these blocks in its cache for efficiency. Quite often, clients will access those blocks in the near future again. Furthermore, the cleaner process will need those blocks soon, as it copies them from the log into the storage area.

LD dynamically determines at run time which data blocks in the in-core segment belong in a log segment and which could go into a direct segment. It does this by determining the category of each of the data blocks in the in-core segment. Although in previous subsections we argue that blocks of the categories *large range of clustered blocks* and *large range of unclustered blocks* can be written as direct segments, for simplicity, we only write blocks of the former category to disk within direct segments in our current prototype.

Our prototype currently uses the following algorithm to determine whether data blocks in the in-core segment are candidates to be written as a direct segment. Data blocks in the in-core segment are written together in a direct segment if the following four conditions are all met.

(1) The blocks belong to the same disk file for which the client has requested clustering.

(2) The blocks have consecutive logical addresses.

(3) The blocks form 256 KB of data.

(4) The blocks start at a logical position in the disk file that is aligned on a 256 KB boundary.

If all these conditions hold, then LD will form one or more direct segments and write these directly into the storage area. If the size of the consecutive data is not a multiple of 256 KB, then the remainder is left in the in-core segment and will eventually be written to disk as part of a log segment.

These rules are more strict than necessary but considerably simplify our algorithm. Especially the last rule is rather strict. However, without this rule implementing data reorganizations would become even more complex than it already is. The choice for these rules also simplifies the design of the internal structure of the storage area, which will be discussed in Chapter 8. Future research is needed to develop a more sophisticated algorithm that better exploits the clustering of the data in the in-core segment.

The size of the in-core segment has been chosen larger than the size of a direct segment in order to increase the chance of actually finding enough consecutive data to fill a direct segment. If the in-core segment is roughly the same size as the direct segment, that is 256 KB plus a little more to hold log tuples, then any concurrency in the write commands to LD from multiple clients would considerably lower the chance of LD finding sufficient consecutive data to fill one direct segment. For instance, consider our old example of Figure 5.2. Two clients each write 256 KB in their own disk file. They both issue write commands of 128 KB each, which are interleaved by LD. Therefore, the in-core segment would be full after writing the first 128 KB of both disk files. As a result, LD cannot form a direct segment even though both clients actually write 256 KB consecutive data in the end.

In our prototype, we have chosen to use an in-core segment of 1 MB which is four times as large as a direct segment. This includes any blocks used to hold log tuples. Therefore, when LD examines a full in-core segment, it can form at most three direct segments from the data in the in-core segment. In this setup, LD can handle our previous example much more efficiently since it can form 256 KB segments from the two 128 KB parts of each disk file. LD can, subsequently, write these segments as direct segments into the storage area as they contain 256 KB of clustered data. Figure 5.5 shows how two direct segments are extracted from the in-core segment in our example. The first direct segment is formed with data from disk file 1, the second direct segment is formed with data from disk file 2. Of course, when possible, LD will write both direct segments together in one physical disk operation, thereby utilizing the maximum available disk bandwidth.

The four original log tuples for the 128 KB parts, which were shown in Figure 5.2(c) are changed when LD forms two direct segments from these 128 KB parts as it combines them two by two. LD must now change these four log tuples, not because they refer to different logical block ranges, but because their phys_addr fields have changed. They must now point to physical addresses within the storage area as the corresponding client data blocks have been written within direct segments. We cannot show the changed log tuples since we have not included the phys_addr field in the figure.

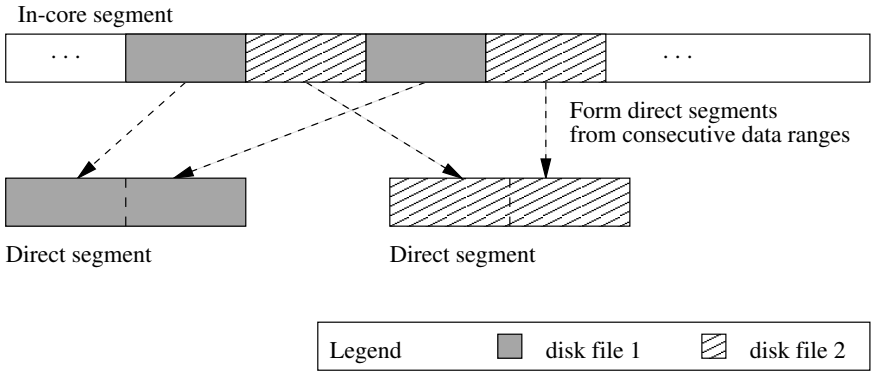


Figure 5.5: Two 128 KB ranges of consecutive data blocks from disk file 1 and two such ranges of disk file 2 in the in-core segment are aggregated into two 256 KB direct segments.

5.7.7 Direct Segments and Maintaining Data Integrity

The problem with writing direct segments is that it may result in data being written to disk out of order. The data blocks in a direct segment are written to disk before data blocks from other commands, which have to wait until the in-core segment is full at which time they are written to disk in a log segment. Consequently, data from commands sent at a later time may reach the disk before data from commands sent to LD at an earlier time. However, one of LD's requirement was that the results of reordering commands may not be visible to the outside world after a crash. Therefore, if, for efficiency reasons, LD writes data blocks to disk in a different order than they have arrived at LD, LD must hide these effects from clients after recovery. In short, LD must recover the disk to a state that incorporates the client's changes in the order they were sent to LD.

In the following example, we look at how LD manages to hide the effects of writing data out of order in direct segments. Our example is depicted in Figure 5.6. Figure 5.6(a) shows incoming write commands to LD. Suppose the gray commands write 256 KB of consecutive data in disk file 1 in two separate calls of 128 KB each. However, between the first and second call a number of other write commands are sent to LD. These commands concern other disk files and clusters. The actual commands are irrelevant for our example. In the figure they are all shown hatched. All commands in our example were sent on one stream, and therefore, LD guarantees that the commands will be recovered in that order after a crash.

Figures 5.6(b) and (c) show the data blocks that were written by these commands in the in-core segment and the corresponding log tuples. The 256 KB of data that are written in disk file 1 are candidates for forming a direct segment since they match the criteria mentioned earlier. Therefore, LD aggregates these data together and writes them to disk in a direct segment, as is shown in Figure 5.6(b). Next, LD updates the still in-core log tuples by changing their phys_addr fields to point to the corresponding locations where the direct segment will be written in the storage area.

Now, let us consider to what state LD should recover if the system crashes at this

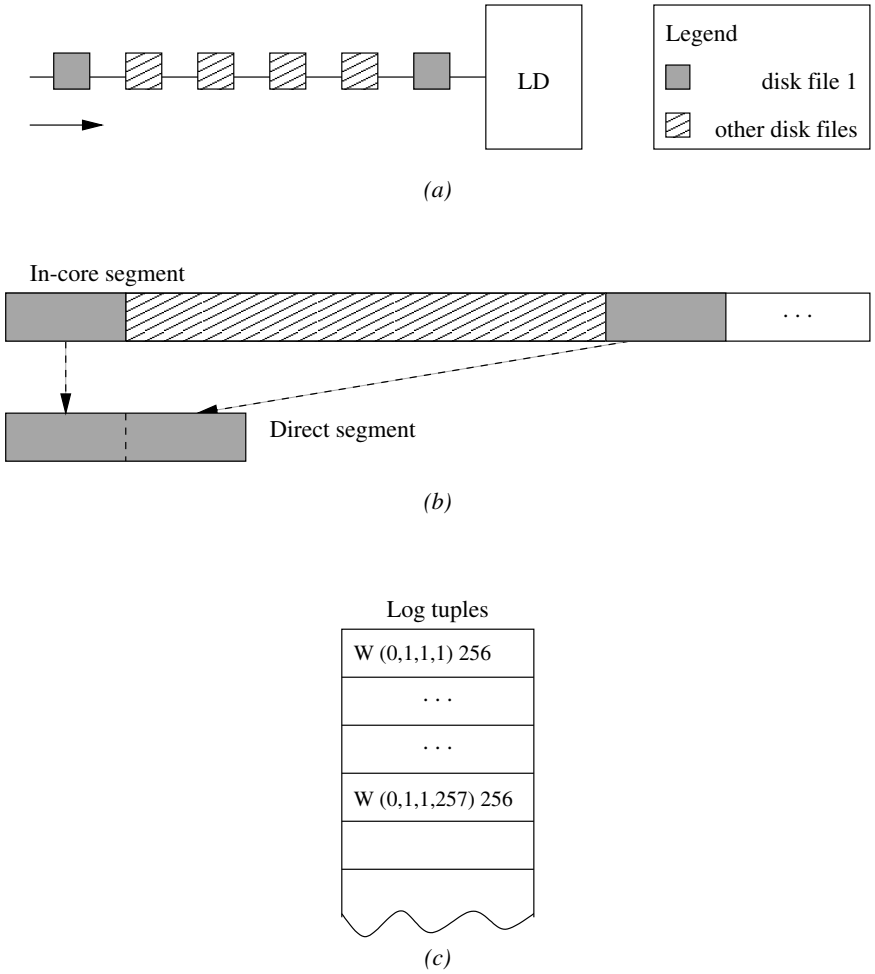


Figure 5.6: Preserving the correct write order after forming and writing direct segments. (a) Two write commands that write consecutive data in disk file 1 arrive at LD with other commands in between. (b) Forming a direct segment with data for disk file 1. (c) The two log tuples of the two write commands.

moment. The data for disk file 1 have already been written to disk in the storage area within a direct segment. However, the in-core segment was not written to disk yet and is, therefore, lost due to the crash. Notice that the command for writing the last 128 KB of disk file 1 has actually been issued *after* the hatched commands in our figure. Therefore, if LD recovers the 256 KB of disk file 1, then LD must also recover the data written by the hatched commands as well, however, that is impossible, because those data were still in the in-core segment, which was lost during the crash. Therefore, disk file 1 may not be recovered, or, at least, the last 128 KB may not be recovered.

In LD, the problem of changing the write order by writing direct segments is avoided by the way LD writes its log tuples. LD writes these log tuples to disk following the rules given in Section 5.6. One of the guarantees that these rules provide is that all in-core log tuples are atomically written to disk. These log tuples determine whether a command will be recovered or not after a crash. Only if the log tuples of the commands that wrote data in disk file 1 are stored on disk, will the entire 256 KB of the disk file be recovered. As a result, when LD writes direct segments to disk, they will not be recovered until the corresponding log tuple is also written to disk as part of a log segment. Of course, every direct segment must be written to disk before a log segment with log tuples referring to that direct segment is written to disk; in fact, this is another one of the rules.

In this particular example, the rules guarantee that if the log tuples of the direct segment, containing data of disk file 1, are on disk, then the log tuples of the hatched commands and their corresponding data blocks are also on disk because they are written together in the same log segment. In this case we also assume that the data of the hatched commands are written in the log segment with the log tuples. Therefore, if the crash happens after writing the in-core segment to disk, then all log tuples had been written to disk, and both disk file 1 as well as the hatched commands can and will be recovered. However, if the crash happened after writing the direct segment, but before writing the in-core segment, then none of the log tuples have reached the disk, and therefore, none of the commands are recovered. In other words, writing a direct segment to disk has no effect on the state to be recovered until the corresponding log tuples are written to disk as well. How LD handles reclaiming the disk space of these data will be discussed in Chapter 7.

5.8 Writing Segments to Disk

In the previous sections we have discussed how newly written data is first accumulated in an in-core segment. While data blocks are in main memory, LD can manipulate them and try to find consecutive ranges of data blocks to write as direct segments. Eventually, all data blocks in the in-core segment are written to disk either in a slot of the log area as a log segment, or somewhere in the storage area as a direct segment. When LD writes a log segment or a direct segment, it may never use in-place updates. Therefore, each new log segment or direct segment must be written into free space in the log area or the storage area, respectively. In this section, we discuss what happens when the data is written to disk.

5.8.1 When to Write a Segment

Logically seen, LD has split client commands that write client data blocks into two operations: first, a large, efficient segment write (either a log segment or a direct segment); second, in some cases, a relatively costly reorganization operation. Additionally, the reorganization operation can be postponed until a later time. Since writing large segments can be done very efficiently, LD can handle periods of heavy disk write traffic. The reason for this is that LD can postpone the reorganizing activity until periods of less disk traffic arrive. The ability to postpone reorganizing activity is an advantage because in practice disk traffic is often 'bursty'. The downside is that the clustering of files that have blocks in the log may temporarily be broken, which may result in reduced read performance.

Over time the in-core segment fills up with client data and log tuples. The exact moment when LD writes a segment of data to disk depends on a number of conditions. There are three conditions that LD checks to determine when it is time to write a log segment to disk. If any one of the following conditions is met, the segment is written to disk.

(1) The contents of the in-core segment (client data blocks and log tuple blocks) have reached the maximum size of 1 MB.

(2) The number of log tuples in the segment has reached a certain maximum.

(3) A certain time interval has passed since a first operation has been stored in the in-core segment.

(4) A client calls `ld_flush`.

If the first condition is true, LD first tries to form and write direct segments to disk, as described in a previous section. This process may create enough free room in the in-core segment, so that LD can postpone writing a log segment. In theory, it is possible that LD does not write a log segment for a very long time. However, since log segments are necessary for recovering a recent state of the disk, it is desirable to write log segments every so often. That is why LD also checks the second and third condition. The second condition assures the clients of LD that after a certain number of operations LD will write a log segment which will make these operations recoverable. Note that this does not guarantee that a client will never lose more than a certain amount of client data because the amount of log tuples provides only an upper bound of the amount of data that is written. However, we have chosen this condition for simplicity in our current prototype. The third condition assures that operations that are older than a certain period of time will also be recoverable. The last condition applies if the client forces LD to flush its buffers to disk by calling `ld_flush` from LD's interface.

The first condition is checked each time that LD tries to add data into the in-core segment as part of executing a client command. In principle, if the amount of data that a client command, such as `ld_write_data`, is writing does not fit in the remaining space of the in-core segment, then LD tries to form direct segments to make more room in the in-core segment. If that does not help, LD can either split said client command into smaller parts so that the first part still fits within the current in-core segment, flush the full segment to disk, and start a new in-core segment. Or LD can flush the in-core segment, which is

not completely full yet, to disk as a log segment, and store all data blocks of said client command in a new in-core segment. In the next section, we will look at how LD can split commands.

Since direct segments are written into the storage area, the layout of direct segments is different from log segments. A direct segment only contains data blocks, it does not hold log tuples nor a log segment header or trailer. In contrast to a log segment, a write of a storage segment does not have to be made atomic with the use of a checksum. A system failure during a write of a direct segment has no influence on the recovery mechanism because the log tuples that correspond to the data in the direct segment will certainly not have been written to disk yet. Therefore, during recovery it is as if the direct segment has not been written at all. The algorithm of allocating space for these segments in the storage area is discussed in Chapter 8.

5.8.2 Freeing Client Data Blocks

As long as LD has not written a log segment to disk, the client data blocks that are in the in-core segment and the corresponding client commands are not recoverable yet. This situation has consequences for the way LD must deal with freeing obsolete client data blocks. Clients can issue commands that delete an old client data block, either by explicitly deleting it or by overwriting it with new contents. In both cases, a log tuple describing the command is stored in the in-core segment. In the latter case, the new contents (also a block) is also stored in the in-core segment, which makes the old version of this block obsolete.

However, if a client issues a command that results in an old client data block to become obsolete, LD cannot simply mark the disk space that was occupied by that old client data block as ‘free’. The problem with this approach is that the command is not yet recoverable on disk: the log segment with the log tuple has not been written to disk. If LD already marks the disk block as ‘free’, LD could reuse its disk space to hold other data, thereby overwriting the old contents of the block. However, if that disk space is reused then LD cannot recover to a correct consistent state if a crash occurs before the log segment is written to disk.

Therefore, LD may not reuse the disk space of client data blocks that have become obsolete due to commands whose log tuples are still in the in-core segment. Only after that log segment has safely reached the disk, can LD mark the disk space of obsolete block as ‘free’ and reusable. The way LD implements this scheme is by storing in a list the disk addresses of client data blocks that have become obsolete since the last log segment write. Those blocks are not marked as ‘free’ in the FreeMap yet. After the log segment is written to disk, LD marks all addresses in this list as ‘free’ in the FreeMap, and clears the list. The problem of freeing obsolete blocks returns in another context in Section 7.9 on page 188 when we discuss the recovery process in LD.

5.9 Splitting Commands

In order to write log segments that are as full as possible, LD can split up commands into smaller parts. The representation of most commands in a log segment is small. Often

only a log tuple is necessary, and sometimes a little amount of header data. Therefore, accumulating such commands in the in-core segment will lead to log segments that are full or almost full before LD writes the log segment to disk. The only exception is a `ld_write_data` client command, which takes up as much space as the data that are being written. Therefore, to be able to fill log segments as full as possible, LD can split up these write commands by breaking up the range of blocks that is written by those commands into smaller parts. Note that splitting up write commands is not only an optimization to write fuller log segments, but is also a necessity if a client writes more data than fits within one log segment in one write command.

For example, suppose LD is almost ready to write a new log segment (1 MB) to disk, but there is still another 100 KB of free space left in the new log segment. Next, a client sends a command to write 200 KB of data in a disk file. LD can now break up that write command into two commands writing 100 KB each. For each of these two commands a new log tuple is generated. These log tuples replace the original log tuple. The first 100 KB is added to the log segment that is about to be written; as a result, a full log segment is written to disk. The second 100 KB will be written to disk in the next log segment.

To uphold the data integrity properties of the original write command, the two smaller write commands must be recovered as a unit; therefore, LD uses its ARU mechanism to assure they belong to a single atomic recovery unit. If a client sends a write command within an already running composite ARU, then LD can simply break that command into smaller write commands without fear of endangering the data integrity. The data integrity of these commands is maintained because the surrounding ARU already guarantees the atomic recovery of all commands within that ARU, including the commands of the broken up write command. However, if a client sends a write command as a simple ARU, then LD starts a composite ARU before it breaks up that command into smaller commands and commits that composite ARU after the last part has been written into the in-core segment. This new ARU is transparent to the user.

For simplicity, LD currently only splits a write command if the amount of data it writes is larger than the size of a direct segment, which is 256 KB. Smaller write commands are not split. This heuristic seems a reasonable choice, since if the in-core segment does not have room enough to hold the data of a small write command, even after forming direct segments, then it can be considered sufficiently filled. Writing such an almost full in-core segment to disk and starting a new empty in-core segment saves LD the trouble to split the command and generating the corresponding log tuples. However, if LD does not split commands that are larger than 256 KB then LD may have to flush in-core segments that have at least 256 KB of free space to disk, which lowers the effective bandwidth utilization of the disk.

5.10 Cleaning the Log

The log area fills up with new log segments over time. At some point, LD must prune the log to create free space in the log area for new log segments. In general, LD prunes the log by cleaning the least recently written log segments. LD can reuse the disk space of a

log segment only if it does not contain data that are still in use, either by a client or LD itself. As mentioned before, each log segment contains two types of data: log tuples and client data.

To prune the number of log tuples in LD's log, which are needed to replay the effects of client commands during recovery, LD periodically makes a *checkpoint*. A checkpoint ensures that the results of all client commands executed thus far are safe on disk and can be immediately recovered after a crash. Consequently, after making a checkpoint, all log tuples in LD's log are not needed anymore. Checkpoints are discussed in detail in Chapter 7.

Making a checkpoint, however, is not sufficient to prune the log completely since the log may contain client data that are still in use by clients. To actually shrink the log, LD must also copy out client data from the log into the storage area. This task is done by separate processes called **cleaners**. The task of these cleaners is twofold. First, they create free space in the log area for new log segments by copying client data blocks from the log to the storage area. Second, the copy process tries to place the client data blocks in the storage area according to their clustering requirements, which was Requirement 3, on page 42. After LD has copied out all client data blocks that are still in use by clients from a log segment to the storage area, and has made a checkpoint, which allows all log tuples in that log segment to be discarded, LD has cleaned this log segment and its disk space can be reused. We will deal with cleaners in more detail in Chapter 8.

Chapter 6

The Metadata Area

The metadata area is the place on disk where LD keeps its internal administration. In Chapter 4, we have already mentioned two data structures that are part of LD's metadata: the Mapping, which keeps track of the locations of the disk files and disk clusters on disk, and the FreeMap, which keeps track of which disk blocks are used and which are free. In this chapter, we look closer at each of these two data structures and explain their design and implementation. Furthermore, we will introduce two more data structures that are necessary to help LD manage the Mapping and FreeMap. Additionally, we present some techniques that allow LD to manage updates to LD's metadata efficiently.

The structure of this chapter is as follows. First, we present in great detail the Mapping. Because of the complexity of the Mapping we have spread its discussion over two sections. Section 6.1 presents the design of the Mapping. In that section, we explain what kind of information is stored in the Mapping, present its interface, and briefly discuss how this interface is used to help implement LD's global interface. Section 6.2 presents how the Mapping is implemented with a variant of a B-tree data structure. We also focus on the method we use to store information in the B-tree compactly. Section 6.3 presents the FreeMap.

Both the Mapping and FreeMap are persistently stored in blocks (metadata blocks) on disk. How LD keeps track of these blocks on disk is the subject of Section 6.4. LD uses two techniques to efficiently write metadata blocks to disk in order to keep the overhead of maintaining metadata low. The first technique is called the 'differential technique', which is presented in Section 6.5. Section 6.6 presents the other technique, which is the 'staccato write'. This chapter ends with discussing how LD deals with keeping metadata and client data mutually consistent on disk. This is important since LD must be able to recover to a consistent state after a crash.

6.1 Design of the Mapping

LD supports a number of storage abstractions, which were introduced in Chapter 3: disk clusters, disk files, logical blocks, cluster headers and file headers. In order to manage these abstractions, LD keeps information about them in the **Mapping**, which we have

already mentioned in previous chapters. The Mapping in LD has a number of purposes. First, the Mapping is used to keep track of which disk files and disk clusters exist on disk. Second, the Mapping is used to store the variable-size header information that clients can associate with a disk file or disk cluster. Last, it is used to find the actual locations of both committed and uncommitted logical blocks of disk files on disk. Every logical block (both committed and uncommitted) has a unique logical block address, whose mapping onto a physical block address is stored in this Mapping.

6.1.1 Committed and Uncommitted Information

Before we present the design of the Mapping, we first look at the types of information that are stored in the Mapping. The information in the Mapping helps LD manage LD's storage abstractions, such as logical blocks, disk files, and disk clusters. We distinguish two types of mapping information: committed and uncommitted mapping information. Committed and uncommitted mapping information are fundamentally different. **Committed mapping information**, or simply **committed information**, refers to metadata that describe the state of logical blocks, disk files, disk clusters, or header information on disk. More concretely, LD keeps committed information for each storage abstraction which tells LD whether the storage abstraction exists (i.e., has been created by a client) or not. If it exists, the committed information also tells LD where to find it on disk. For cluster and file headers, the committed information contains the actual contents of the header. Committed information is the basic information that LD needs to keep track of the data that clients store using LD's storage abstractions.

The second type is **uncommitted information**, which refers to *changes* to logical blocks, disk files, disk clusters or header information on disk. Examples of such changes are a creation of a new disk file, an (over)write of a logical block, a deletion of an existing logical block, or a write of a file header. If these changes originate from commands executed within ARUs, they result in uncommitted data. Recall that ARUs allow clients to group multiple commands into a single atomic action. Changes made within an ARU are only tentative; they will become definitive when the ARU commits. Therefore, until the ARU commits, these changes must be stored separately in the Mapping because they cannot be stored as committed information in the Mapping yet.

Also recall that clients can send a command outside an ARU, which we call a simple ARU. Conceptually, a simple ARU is a special case of a composite ARU; it is a short running composite ARU with only one command which is immediately followed by a commit. Therefore, every updating client command can be considered being sent within an ARU: either a simple ARU or a composite ARU. This simplifies our discussion because we only have to deal with updates being sent within an ARU. However, in an actual implementation simple ARUs will, in general, not be implemented as composite ARUs that are immediately committed. In that case, simple ARU commands will directly change committed information instead of first generating uncommitted information that will be converted into committed information after the commit. However, in the following discussion we will focus on changes made within ARUs.

Each ARU can be divided into three phases:

(1) **Start ARU** - the beginning of the ARU, which is started implicitly in the case of

a simple ARU, and explicitly in the case of a composite ARU, by a client calling `ld_start_aru`.

(2) **Execute command(s)** - the client command(s) sent within the ARU are executed. In a simple ARU, only one command is given by a client; within a composite ARU, multiple commands may be given by a client. The execution of these commands generates uncommitted data and, thus, results in changes to the Mapping.

(3) **Commit ARU or abort ARU** - the end of the ARU, which can be committed or aborted. A simple ARU commits implicitly, and a composite ARU is explicitly committed by a client calling `ld_commit_aru`. This action turns the uncommitted changes made by commands in the previous phase into committed information. An ARU can be aborted explicitly by a client calling `ld_abort_aru`, which aborts all uncommitted changes made by commands in the previous phase.

The changes made within an ARU (i.e., uncommitted data) are not visible to others until that ARU commits. Committed data, on the other hand, are visible to all ARUs. Therefore, each ARU can see the committed versions of all logical blocks, disk files, and disk clusters, as well as the uncommitted changes that the ARU itself has made to them, which are private to that ARU. Conceptually, changes made to any data abstraction within an ARU create a new private copy of the data abstraction and are, therefore, invisible to others. After a commit of an ARU, the private copies become the committed versions, so that other ARUs can also see the changes made by that ARU.

LD implements this scheme as follows. LD has one version of the committed information, which is visible to all ARUs and describes all committed storage abstractions. Furthermore, for each running ARU, LD keeps a list of all uncommitted changes that are made within that ARU. These changes reflect updates to the state of storage abstractions on disk, but are not immediately applied to the actual committed information. The information in these lists is called *uncommitted information*. Together, the committed information and the uncommitted information, which LD holds for each running ARU, make up the mapping information in LD. When LD consults the mapping information, LD first consults the private list of uncommitted changes of the ARU before looking in the committed information. As a result, commands sent within an ARU see all the changes that were previously made within that same ARU. When an ARU commits, the uncommitted changes in its private list are applied to the committed information and become, therefore, visible to other ARUs.

As an example of how LD uses committed and uncommitted information, suppose that LD receives a read request for a certain logical block within a running ARU. During the execution of this command, LD must determine whether this block exists, and if so, it must know where it is located on disk. First, LD checks the list of changes (i.e., uncommitted information) for that ARU to see if the requested block has already been changed within that ARU. If it has, there will be an entry in the list describing the change. The fact that the block has been changed could either mean that the block has been deleted or that a new version of the block has been written. If it has been deleted within this ARU, LD reports this back to the client. If it has been rewritten, the entry in the list tells LD where on disk it can find the new version of the block. Subsequently, LD reads the block from

disk and returns it to the client. If the block has not been changed within this ARU, then LD consults the committed information which LD holds on all its committed storage abstractions. The committed information reveals to LD whether the block exists and if so, it provides LD the block's disk address. After reading the block from disk, LD returns it to the client completing the client's read request.

We can make another division in the meta-information that LD stores: logical block information and header information. **Logical block information** is the mapping of logical block addresses to physical block addresses. **Header information** are the file headers and cluster headers. We make this distinction because both types of information have different physical properties: logical block information consists of fixed-size data items (physical block addresses), whereas header information consists of variable-size data items. Consequently, they need to be managed differently. This division is orthogonal to the committed and uncommitted division. Therefore, for both block information as well as header information, committed state information and uncommitted change information is stored.

Table 6.1 sums up the different kinds of meta-information that LD stores. For each logical block or header information, LD keeps committed information on whether it exists or not. The existence of the header information, indirectly, tells LD whether the corresponding disk file or disk cluster exists on disk, that is, whether they have been created by a client. Recall that a header consists of two parts: a *private* part, which is used by LD, and a *public* part, which is used to store data of clients. Whenever a client creates a disk file or disk cluster, LD fills the private part of the header of that disk file or disk cluster. The public part of the header is initially left empty (see Section 4.2 on page 79).

Table 6.1: Possible values for committed and uncommitted meta-information in the Mapping.

Committed Information		**Uncommitted Information**	
Block	**Header**	**Block**	**Header**
nonexistent	nonexistent	unchanged	unchanged
existent	existent	new block	new header
		block deleted	header deleted

The uncommitted information for the storage abstractions has three possible values: the storage abstraction is unchanged, it has been rewritten, or it has been deleted. For logical blocks, the interpretation of these changes are straightforward. However, for headers the interpretation of a deletion needs a little explanation. LD uses the presence of (the private part of) a header to indicate whether the associated disk file or disk cluster exists. Therefore, a header, including both its private and public part, can only be deleted by deleting the corresponding disk file or disk cluster. A client can delete the public part of a header by writing an empty header into the public part. In our table such an action falls under the category of writing a 'new header'.

6.1.2 Storing Information in the Mapping

LD could store committed and uncommitted information in two separate data structures. However, we have chosen to store both types of informations in one data structure: the Mapping. The reason for this is that it is not very difficult to encode both types of information in one data structure, and maintaining one data structure is simpler than maintaining two.

The Mapping associates with each logical block or header one of five different values, which are listed in Table 6.1. As mentioned before, the Mapping can be seen as a collection of (key, value)-pairs. The key indicates the block or header on which information has been stored in the value field. Furthermore, the key also indicates whether the value field refers to committed or uncommitted data, that is, whether it contains committed information or uncommitted information. The corresponding value field stores the actual mapping information for the block or header. We will explain how the information is stored in the Mapping in detail below.

Storing Committed Information

Table 6.2 lists how the different types of values are stored in the Mapping. Let us first look at the committed values. The way LD encodes that a block or a header does not exist is simple: store no information about it in the Mapping. All existing blocks and headers combined only occupy a small amount of the total available key space; therefore, it is more space efficient to simply leave out the entries for blocks and headers that do not exist.

Table 6.2: (key, value)-pairs in the Mapping. Keys consist of logical addresses of the form (ARU_id, cluster_id, diskfile_id, offset), which has been abbreviated to (A,C,D,F) in the table. The value field is either a physical block address, a piece of header data, or a special NIL value.

Mapping information	**Block**		**Header**	
	Key	Value	Key	Value
Committed				
nonexistent	-	-	-	-
existent	(0,C,D,F)	phys_addr	(0,C,D,0)	header
Uncommitted				
no change	-	-	-	-
new version	(A,C,D,F)	phys_addr	(A,C,D,0)	header
deleted	(A,C,D,F)	NIL	(A,C,D,0)	NIL

For each block that does exist, a (key, value)-pair is stored in the Mapping. The key is the internal logical block address of that block, and the value field is the actual physical block address where that block can be found on disk. Recall that an internal logical block address is a tuple (aru_id, cluster_id, diskfile_id, offset). The last three fields uniquely

identify a block in a disk file within a cluster. The aru_id indicates whether the block is committed (aru_id is 0) or uncommitted (aru_id > 0). In the latter case, the aru_id indicates the ARU that has done something with that block.

The committed information for existing headers is also a (key, value)-pair. The key indicates the logical address of the file header or cluster header, and the value field holds the actual header information. We assume for the moment, that the size of the header is also stored within the value field. Later on we will see that the size is actually stored elsewhere. The aru_id field in the key is 0 because this entry represents committed information about the header. The offset field is 0 to indicate that the key refers to a header and not a logical block. The cluster_id and diskfile_id indicate the disk file or cluster, whose header is stored in that entry.

Storing Uncommitted Information

Uncommitted information has three different types of values: unchanged, a new version, or deleted. For blocks and headers that have not been changed in an ARU, no explicit uncommitted information is stored in the Mapping. If a block is written within an ARU, then the key indicates the logical address of the block that has been written. The aru_id in the key indicates which ARU has written it. The value field is the new physical block address of the new version of the block.

If a new header has been written, the key indicates the disk file or cluster, whose header has been written (offset is 0), and in which ARU it has been written (ARU > 0). Note that when a new header is written in the Mapping, this could mean that a client has created a new disk file or disk cluster or that a client has changed the public part of a header. In both cases, the header information is changed. The value field of the (key, value)-pair is the actual contents of the header.

The representation of a deletion of a block or header in the Mapping uses the same key format as was used to describe writing a new block or header. However, the value that is stored in the Mapping is a special NIL value. This indicates that the corresponding block or header has been deleted. Note that we cannot represent deleting a block or header by leaving the entry out of the Mapping because then LD cannot distinguish whether a block or header has been deleted within an ARU or whether it has not been changed. Therefore, the Mapping has a separate representation for deletions within an ARU.

6.1.3 Interface of the Mapping

Now that we have discussed what kind of information the Mapping should keep, let us turn to its interface. The interface is actually relatively simple. Conceptually, the Mapping is a table holding (key, value)-pairs, and the interface of the Mapping should allow new pairs to be stored in the Mapping. Additionally, it should allow existing pairs in the Mapping to be updated, retrieved (fetched), or deleted. Listing 6.1 shows these basic operations on the Mapping.

Conceptually, the Mapping has two sets of operations: one set to manipulate headers, and the other set to manipulate disk addresses in the Mapping. `ldmap_fetch_hdr` and `ldmap_fetch_addr` are used to query the Mapping. The caller supplies the key, consisting

```
/* Fetch, store, or delete a header from the Mapping. */
int ldmap_fetch_hdr(uint32_t aru, uint32_t cluster, uint32_t file,
                    uint32_t fetchsize, char *data);
int ldmap_store_hdr(uint32_t aru, uint32_t cluster, uint32_t file,
                    uint32_t storesize, char *data);
int ldmap_delete_hdr(uint32_t aru, uint32_t cluster, uint32_t file);

/* Fetch, store, or delete an address range from the Mapping. */
int ldmap_fetch_addr(uint32_t aru, uint32_t cluster, uint32_t file,
                     uint32_t offset, uint32_t count, uint32_t *addrs);
int ldmap_store_addr(uint32_t aru, uint32_t cluster, uint32_t file,
                      uint32_t offset, uint32_t count, uint32_t *addrs);
int ldmap_delete_addr(uint32_t aru, uint32_t cluster, uint32_t file,
                       uint32_t offset, uint32_t count);
```

Listing 6.1: Fetch, store, or delete information from the Mapping.

of the aru_id, cluster_id, file_id, and offset. The function retrieves the requested header or physical address associated with the key, if present. If it is not present, an appropriate error is returned. The latter function can retrieve the physical addresses for a whole consecutive range of logical addresses in the buffer `addr`, which must be supplied by the caller. When requesting a range of physical block addresses, the function `ldmap_fetch_addr` puts a special EMPTY value in the buffer for each key (i.e., logical block address) in the range that has not been stored in the Mapping. For committed information (i.e., the function has been called with aru_id 0), this EMPTY means that the block does not exist. For uncommitted information, this value means that the block has not been changed in the ARU.

`ldmap_store_hdr` and `ldmap_store_addr` are used to insert new entries or to modify existing entries. Likewise, `ldmap_delete_hdr` and `ldmap_delete_addr` delete entries from the Mapping. Note that an easy way to delete all physical addresses of a particular disk file from the Mapping is to call the function `ldmap_delete_addr` where its arguments are as follows: the key is the first block in the disk file (offset 1) and the `count` is the maximum possible offset.

Additionally, the interface of the Mapping supports functions to find existing entries in order. These functions are akin to the functions in Listing 3.2 in Chapter 3, such as `ld_next_unused_cluster`, which were used to search for used or unused diskfile_ids, cluster_ids, or offsets. The corresponding functions in the Mapping are `ldmap_next_entry` and `ldmap_prev_entry`, which are given in Listing 6.2. Given a key, these functions will return the following or previous (key, value)-pair that is actually stored in the Mapping.

6.1.4 Using the Mapping Functions

In this subsection, we will briefly sketch how the Mapping functions presented above are used during the execution of LD's main interface functions, described in Chapter 3. Most

```
/* Find the next or previous entry within the Mapping. The search forward or
 * backward starts from a given start point. If found, the answer is stored in the
 * parameters nclusterp, nfilep and noffsetp. Otherwise, an error is returned. */
int ldmap_next_entry(uint32_t aru, uint32_t cluster, uint32_t file,
                     uint32_t offset, uint32_t *nclusterp,
                     uint32_t *nfilep, uint32_t *noffsetp);
int ldmap_prev_entry(uint32_t aru, uint32_t cluster, uint32_t file,
                     uint32_t offset, uint32_t *nclusterp,
                     uint32_t *nfilep, uint32_t *noffsetp);
```

Listing 6.2: Find previous or next entries in the Mapping.

of these functions require some sort of interaction with the Mapping. Table 6.3 sums up how the Mapping functions are used to implement some of LD's more important interface functions.

The first column of the table lists the functions of LD. For clarity, the functions of LD are listed in groups, separated by horizontal lines in the table. Functions in the same group call the same Mapping functions during execution. The second column shows which Mapping functions are used. Most entries in this table are straightforward. For example, to create a new disk file or cluster, a new header must be created in the Mapping since the presence of a header indicates that the corresponding disk file or cluster exists. Therefore, the function `ldmap_store_hdr` is called. This same function is also necessary to modify an existing header. To fetch a header, the function `ldmap_fetch_hdr` is used. To read, write, or delete client data blocks, their corresponding disk addresses must be fetched, stored, or deleted, respectively.

The penultimate group in the table concerns deleting a whole disk file or cluster. Deleting a disk file is implemented by deleting its header and all of its data blocks. Therefore, both `ldmap_delete_hdr` and `ldmap_delete_addr` are used to remove the corresponding information from the Mapping. In order to delete an entire cluster, this must be done for all disk files in that cluster, and the cluster header must be deleted as well. To find out which disk files are within a cluster, the function `ldmap_next_entry` can be used. The last entry concerns committing an ARU. The steps that are necessary in the Mapping to commit an ARU have already been explained in Section 4.1.2 on page 67. In short, the following steps must be done atomically. First, all changes made within that ARU must be looked up in the Mapping, which can be done with the help of the function `ldmap_next_entry`. Then, any committed entries that are made obsolete by the commit, must be removed from the Mapping. To complete the commit, committed versions of the ARU entries are stored in the Mapping, when appropriate, and the ARU entries are removed. All these steps together require the use of all available Mapping functions.

Conceptually, the above listed functions are sufficient for the Mapping to function correctly. For efficiency reasons, however, other functions may be added in an actual implementation of the Mapping. For example, deleting all information of a cluster in the Mapping can be more efficiently implemented by a separate `ldmap_delete_cluster` function. Without such a function the function `ldmap_next_entry` must be used to find which disk files are present in the cluster to be deleted, which is quite inefficient. A similar

Table 6.3: Using Mapping functions to implement LD functions.

LD function	Mapping function(s)	Comments
ld_create_file ld_create_cluster ld_set_fh ld_set_ch	ldmap_store_hdr	Creating a new disk file or cluster requires the same actions as storing a header of a disk file or cluster.
ld_get_fh ld_get_ch	ldmap_fetch_hdr	Retrieving a file header or cluster header fetches the header from the Mapping.
ld_read_data	ldmap_fetch_addr	Reading a range of blocks.
ld_write_data	ldmap_store_addr	Writing a range of blocks.
ld_delete_blocks	ldmap_delete_addr	Deleting a range of blocks.
ld_delete_file ld_delete_cluster	ldmap_delete_hdr ldmap_delete_addr ldmap_next_entry	Deleting a disk file deletes its header and all its blocks. Deleting a cluster deletes all its disk files.
ld_commit_aru	ldmap_next_entry ldmap_fetch_hdr ldmap_fetch_addr ldmap_store_hdr ldmap_store_addr ldmap_delete_hdr ldmap_delete_addr	Committing an ARU, fetches all uncommitted entries of that ARU in the Mapping, stores committed versions of them in the Mapping, then deletes them.

case can be made for `ld_commit_aru`. Accessing each entry of the ARU separately is not very efficient, a specialized function can perform the same actions more efficiently.

6.2 Implementation of the Mapping

The Mapping contains important metadata about client data on disk; therefore, the Mapping must itself be stored persistently. We have chosen a straightforward option to store the metadata in the Mapping on the same disk as the client data. The Mapping is implemented using a variant of the *B-tree* data structure [Bayer and McCreight, 1972; Comer, 1979]. B-trees and their variants are well-known data structures, especially in the field of databases.

The reason for choosing a B-tree variant is threefold. First, the B-tree is a dynamic data structure, which means that it can dynamically grow and shrink over time. Second, the B-tree uses an efficient index to make lookups in the Mapping fast. Third, a B-tree is order preserving, which means that it stores the mapping information on disk in a way that respects the order of the keys, and therefore, the block information for disk files will remain clustered. Clustering is important since access to a client's disk blocks is often sequential, and therefore, clustering the block information in the Mapping leads to fewer disk accesses to read this block information from disk. This last reason prevents us from using a hash based data structure to implement the Mapping because such a structure is not order preserving.

The B-tree that is used to implement the Mapping consists of a number of nodes that are linked with pointers in a tree shape. Each node is stored on disk in a **logical metadata block**. A logical metadata block differs from logical client data blocks in a number of ways. First, the former hold LD's metadata, while the latter hold client data stored in disk files. Second, the size of each logical metadata block is 4 KB, whereas a client data block is 512 bytes. The reasons for the difference in size are discussed in Section 6.4.3. Each metadata block is identified by a unique **logical metadata block address**, which differs from a client logical block address, which was introduced in Chapter 3. Consequently, we use two separate address spaces, one for client blocks and the other for metadata blocks, which both map onto physical addresses on disk. We will come back to logical metadata blocks and their addressing in Section 6.4.

In our implementation we have used a *W-tree* [de Jonge and Schijf, 1990], which is a variant of the B-link tree [Lehman and Yao, 1981]. Figure 6.1 shows an example W-tree. In a W-tree, the actual data are stored in the nodes on the bottom level of the tree. Nodes in higher levels only serve as an index to speed up finding information in the bottom level. The lowest level is called the **sequence set**. Each higher level is an index on the level beneath it; therefore, the upper levels form a set of indices, called the **index set**. To be precise, this set forms a multilevel index. Conceptually, each node in the sequence set consists of an array of (key, value)-pairs, whereas each node in the index set consists of an array of (key, pointer)-pairs. The pairs within each level and within each node are sorted on the key.

The pointers in an index node are references to nodes at the level directly below. The top level of the tree consists of only one node: the root. The nodes at each level are

also linked from left to right. This construction makes accessing all data at one level in the tree easy, which helps to implement concurrent access to the tree in the future (see Section 6.2.8).

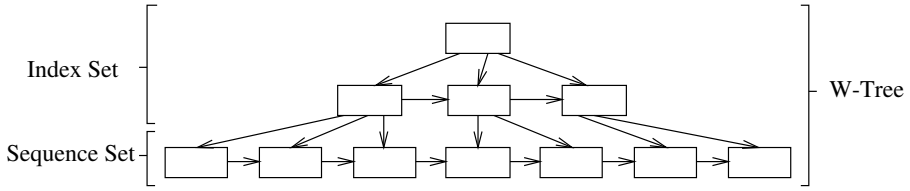


Figure 6.1: A W-tree consists of an *index set* and a *sequence set*. The information in one layer is an index for the information in the layer beneath it.

The structure of the W-tree is such that all keys stored in the subtree pointed to by a (key, pointer)-pair in a node of the index set, are smaller than or equal to that key. This way, to find the value associated with key k in the tree, one starts at the root and examines the (key, pointer)-pairs in that node to find the pair that has the smallest key that is larger than or equal to key k. Then the search continues in the node pointed to by that pointer, until the sequence set is reached where the desired (key, value)-pair can be found.

In the following subsections, we look in more detail how the Mapping is stored in a W-tree. First, we look at how the (key, value)-pairs (i.e, the mapping information) are stored within the tree. To minimize the amount of space needed, LD uses compression techniques to store (key, value)-pairs compactly in structures called **Mapping Parts**. The entire Mapping, then, logically consists of a number of Mapping Parts, each of which stores mapping information. Mapping Parts are discussed in Section 6.2.1 and their use is further described in Sections 6.2.2 through 6.2.5. These Mapping Parts are stored in the nodes of the sequence set of the tree. This is described in Section 6.2.6, which is followed by a discussion of the index set in Section 6.2.7. The last subsection describes some concurrency issues in the tree.

6.2.1 Mapping Parts

Above we have shown that the Mapping associates five types of information with logical blocks and headers. These types were summarized in Table 6.1: two types of committed information (nonexistent, existent) and three types of uncommitted information (unchanged, new value, deleted). Conceptually, the information in the Mapping is stored as (key, value)-pairs. However, in practice, we store the information in a more compact form. For example, let us take a look at how the Mapping stores the committed block address information for a particular disk file. This information is needed so that clients can find the corresponding physical block address given a particular logical block address of a block of that disk file.

Suppose that disk file has diskfile_id (5, 3), and consists of 16 blocks. The 16 logically consecutive blocks within the disk file start at offset 1. The logical block address of the

first block is thus the tuple (0, 5, 3, 1), the address of the second block is (0, 5, 3, 2), and so on. Let us, furthermore, suppose that the corresponding physical blocks are mostly scattered randomly across the disk in small groups. For example, the first four blocks are located at physical addresses *320*, *321*, *322* and *323*. The next three are randomly placed at *502*, *476* and *872*, etc. This situation is depicted in Figure 6.2.

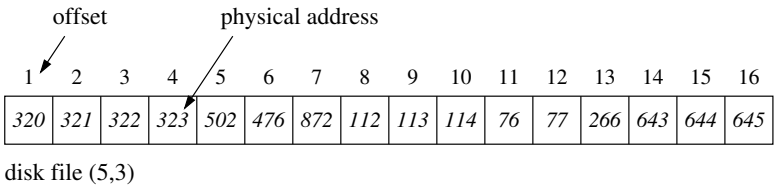


Figure 6.2: Disk file with diskfile_id (5, 3) consisting of 16 blocks. The logical blocks are represented by boxes. For ease of reference, the offsets of the blocks within the disk file are printed above each block. The physical addresses of the blocks of the disk file are printed within the blocks themselves.

The committed block information of this disk file could be stored in the Mapping as 16 individual (key, value)-pairs: ((0,5,3,1), *320*), ((0,5,3,2), *321*), . . . , etc., which is depicted in Figure 6.3. However, this representation contains a lot of redundant information. In the range of keys, the only thing that changes is the offset, which changes in increments of 1. Therefore, we can store the same information in a more compact form. Instead of storing each (key, value)-pair separately, we only store the logical start address, the number of consecutive logical addresses, and the corresponding physical addresses. Figure 6.4 illustrates this method. Using this method we are able to store the same information in a more space efficient way. We call such a construction a **Mapping Part**, or MP for short.

A Mapping Part stores the physical addresses of a *consecutive* range of *logical* addresses within one disk file. It is made up of two parts: the **MP-Header** and the **MP-Data**. The MP-Header includes the logical address of the start of the consecutive range and the length of the range in logical blocks. In short, it describes which part of which disk file is described by that Mapping Part. For clarity, we represent the MP-Headers as a box with a gray background in figures. The MP-Data part contains a list of the physical addresses that correspond to the logical block addresses in the range described by the MP-Header. The exact contents of both the MP-Header and the MP-Data are explained in Section 6.2.6.

It is not necessary that a disk file is represented by just one Mapping Part. Multiple Mapping Parts may be used to describe one disk file. Each Mapping Part then describes a different range of the disk file. We will encounter examples in which the mapping information of one disk file is stored in multiple Mapping Parts later on in this chapter.

Mapping Parts form the building blocks of our Mapping. In the above example, we have used a Mapping Part to efficiently store committed block information. However, Mapping Parts are used to store all types of information listed in Table 6.1. In other words, Mapping Parts are used for block and header information, both committed and uncommitted.

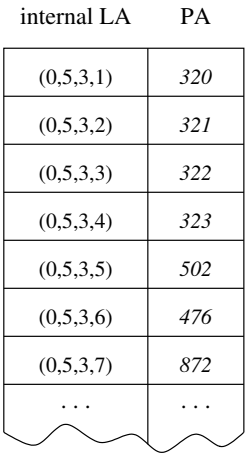


Figure 6.3: The mapping information for the disk file of Figure 6.2. Conceptually, the mapping information is a table of (key, value)-pairs.

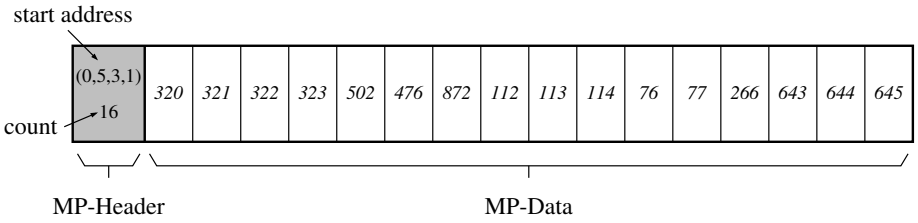


Figure 6.4: A Mapping Part, containing the mapping information for the disk file of Figure 6.2.

6.2.2 Holes in Disk Files

A disk file is a sparse array of logical blocks. A disk file, therefore, does not have to be one large consecutive array of logical blocks. It may contain *holes* anywhere within the disk file. These holes are similar to the holes in UNIX files, which can be created by seeking to a position past the current size of a file, and then writing a block at that position. A hole simply means that at a certain offset within the disk file there are no associated physical blocks.

In mapping terms, a hole in a disk file means that the Mapping simply should not associate a physical address with that logical block address. However, for efficiency reasons, LD sometimes also stores nonexistent blocks explicitly in the Mapping by storing a special NIL value. Therefore, there are two methods to represent a hole in the Mapping:

(1) Storing the special NIL value for the corresponding logical address.

(2) Storing nothing about that logical address in the Mapping.

We call the first method *explicitly* storing holes, the latter is called *implicitly* storing holes.

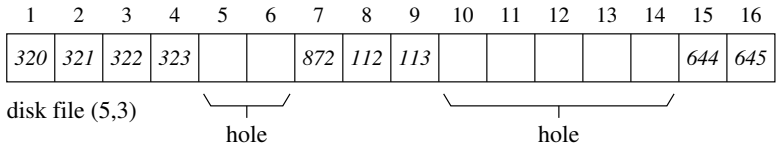


Figure 6.5: A disk file with two holes.

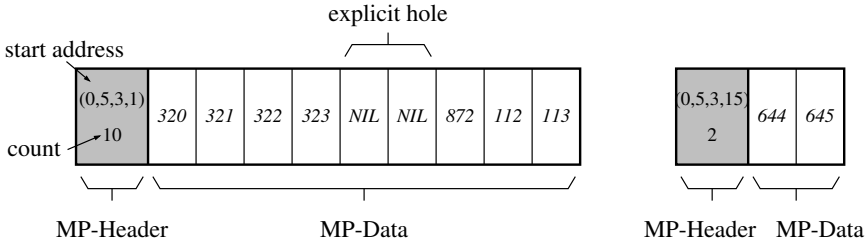


Figure 6.6: Two Mapping Parts describing a disk file with two holes.

We will use the disk file depicted in Figure 6.5 to illustrate both methods. The disk file in Figure 6.5 contains two holes. The first hole is located at logical offsets 5 and 6. The other hole starts at offset 10 and stretches for 5 blocks. Figure 6.6 shows how the mapping information of this disk file can be represented in the Mapping. The first hole is represented *explicitly* by storing two NIL values at logical addresses (0, 5, 3, 5) and (0, 5,

3, 6). The second hole is represented *implicitly*: for logical addresses (0, 5, 3, 10) through (0, 5, 3, 14) there is simply no information being stored. This example also shows us a disk file that uses more than one Mapping Part to represent its mapping information. The first Mapping Part describes the mapping information for logical addresses (0, 5, 3, 1) through (0, 5, 3, 9) and the second Mapping Part holds the information for the addresses (0, 5, 3, 15) and (0, 5, 3, 16).

The representation of the disk file shown above is certainly not the only way to represent that disk file. The second hole could also have been represented explicitly or the first hole could have been stored implicitly or we could have used even more Mapping Parts to store the mapping information, etc. All these representations hold the same information and are thus logically equivalent. However, the shown configuration is the most space efficient. In general, it is more space efficient to represent large holes implicitly and to store explicit NIL values for small holes. The next section will introduce additional methods that allow us to represent mapping information in an even more compact form.

6.2.3 Compression Methods

The Mapping Part presented in the previous section already uses a compact form to store (key, value)-pairs. For a consecutive range of (key, value)-pairs only one key (i.e., the key of the start of the consecutive range of addresses) is stored in a Mapping Part. There are, however, cases in which we can compress the mapping information even further, using other representations. Here, we present two additional methods to store mapping information in a Mapping Part. In total, there are three ways to store values associated with a consecutive range of logical addresses in a Mapping Part: *Enumeration* (**E**), *Sequence* (**S**) and *Repetition* (**R**). The MP-Header contains a type field which denotes which of the three compression methods is used for that particular Mapping Part.

- **Enumeration (E)** — start logical address, length, physical addresses
  With this method each physical address of the range denoted by the MP-Header is listed separately in the MP-Data. If the MP-Header describes a range of N blocks, then the MP-Data contains an array of N physical addresses.

- **Sequence (S)** — start logical address, length, start physical address
  With this method the physical addresses in the range denoted by the MP-Header also form a range of consecutive addresses. If the physical blocks of a disk file are clustered on disk in order, then not only the logical addresses but also the corresponding physical addresses are consecutive. Just as the MP-Header denotes a consecutive range of logical addresses as a start address + length, analogously, we can denote the consecutive range of physical addresses by only storing the first physical address + length. Of course, the length needs to be stored only once. In short, this method only stores one physical address in the MP-Data part, which indicates the start address of the range of blocks referred to by the MP-Header. This method is very space efficient in representing the mapping information of a long range of clustered blocks on disk. This method of compression is similar to the use of extents in extent-based file systems and *data runs* in the NTFS file system [Solomon and Russinovich, 2000].

- **Repetition (R)** — start logical address, length, value
  With this method the physical addresses in the range denoted by the MP-Header all have the same value, which is listed in the MP-Data. The MP-Data, thus, only contains one value. This method is useful to store a range of nil values in the Mapping. A nil value can be used to represent committed information as well as uncommitted information. In the former case, it is used to represent a *hole* in a disk file (see also Section 6.2.2). In the latter case, it is used to represent a deletion of a range of blocks. This compression method is currently not meant to represent actual physical addresses because LD does not support sharing of physical blocks.

We have already seen a typical use of the **Enumeration** compression method in Figures 6.4 and 6.6. Figure 6.7 shows an example of the **Sequence** compression method. Consider a disk file with diskfile_id (5, 4) which has 16 consecutive blocks starting at physical address *320*. Figure 6.7(a) shows how the committed block information of this disk file is stored using the Enumerate method. Figure 6.7(b) shows the more compact Sequence method.

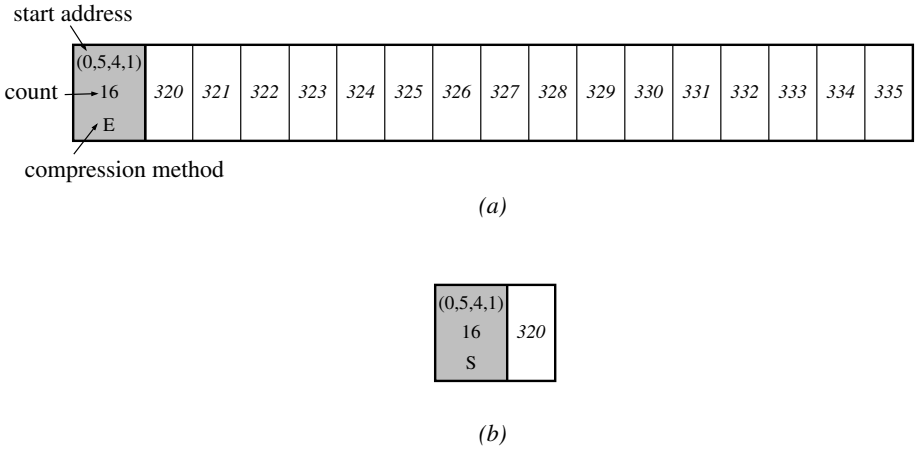


Figure 6.7: An example of the use of the *Sequence* compression method. (a) Mapping information stored with the Enumeration compression method. (b) The same mapping information stored with the Sequence compression method.

Figure 6.8 demonstrates the use of the **Repetition** compression method. Suppose that a client deletes the first 10 blocks of disk file (5,4) within ARU 1. This deletion is stored as uncommitted information in the Mapping. As we presented in Table 6.2, deletions are represented by the special NIL value in the Mapping. Therefore, this deletion operation would associate the NIL value with the corresponding addresses of the deleted logical blocks. Figure 6.8(a) shows how this would be represented in the Mapping with the Enumeration compression method. However, the Repetition compression method allows us to store this more compactly, which is shown in Figure 6.8(b). For completeness, note that the deletion above is still uncommitted. After the commit of the ARU, that

deletion will result in the deletion of the committed blocks and will be represented in the Mapping by removing the committed information for those blocks. Concretely, this will result in deleting the Mapping information for the logical blocks with addresses (0,5,4,1) through (0,5,4,9). The other use for the Repetition method is to store large holes explicitly. However, it is more space efficient to store such large holes implicitly.

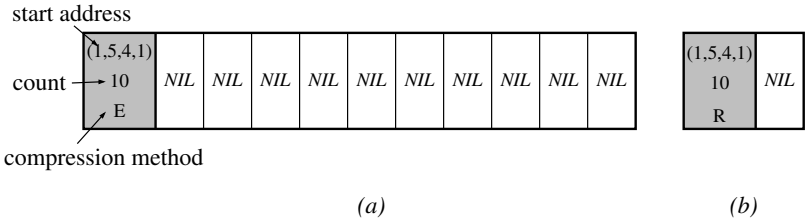


Figure 6.8: An example of the use of the *Repetition* compression method. (a) A deletion of the first 10 blocks of a disk file represented by the Enumeration compression method. (b) The same deletion represented by the Repetition compression method.

A disk file can be represented by multiple Mapping Parts, each of which describes a different range within the disk file and may use a different compression method. LD is free to decide which compression method to use when storing committed or uncommitted information. It can use the best combination to reduce the space needed for the Mapping; or it can use a simpler approach and use the most convenient way to store data, thereby avoiding the time penalty that is involved in trying to analyze which is the best way to store the information.

6.2.4 Disk File Headers and Disk Cluster Headers

Recall that a disk file header and a disk cluster header are variable-size pieces of data that can be associated with a disk file and a disk cluster, respectively. These headers are also stored in the Mapping with the use of Mapping Parts. The header data are stored in the MP-Data part of a Mapping Part. The logical address in the key in the MP-Header is a (aru_id, cluster_id, diskfile_id, offset) tuple in which the offset is 0 (which denotes that this Mapping Part refers to a disk file or cluster header) and the cluster_id and diskfile_id refer to the disk file or cluster to which those header data belong. The length of the header in bytes is stored in the count field in the MP-Header and the actual contents of the header itself, comprising the private and public part, are stored in the MP-Data part of the Mapping Part. The header of a single disk file or cluster must always be contained within one Mapping Part and cannot be spread across multiple Mapping Parts, unlike block information. To summarize, a Mapping Part can either hold header data of one disk file or disk cluster, or block information, but never a combination of both.

Recall that a header has two parts: a private and a public part. For every existing disk file or cluster, LD keeps its own internal meta-information in the private part of the header. Therefore, by looking if the Mapping contains a corresponding header, LD can test for the existence of a disk file or cluster. To the Mapping, however, the fact that a

header has two parts is of no interest. The Mapping does not offer special functionality to manipulate the two separate parts individually. The Mapping considers a header simply as one variable-size piece of data. However, since the private part has a known fixed size, LD can access both parts of the header individually when it receives the whole header from the Mapping. Therefore, in the rest of this chapter, we will not refer to the existence of this division again.

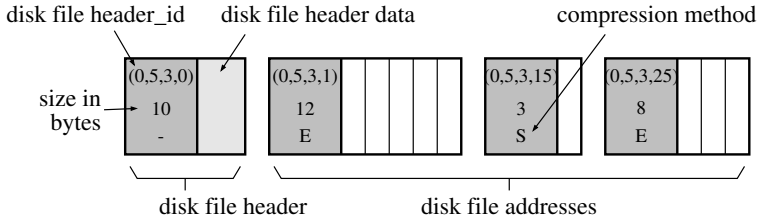


Figure 6.9: Committed information of a disk file, including its disk file header.

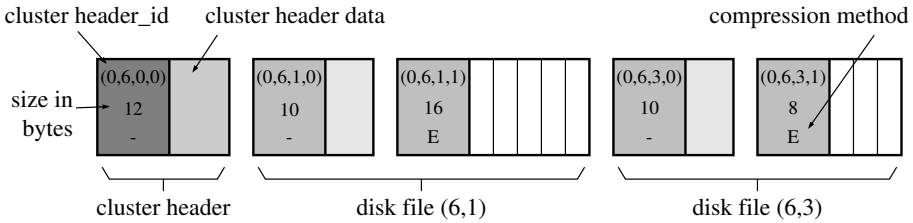


Figure 6.10: Committed information of a cluster with a cluster header. The cluster contains two disk files.

Figure 6.9 shows us how a disk file is represented in the Mapping. Each disk file has a disk file header, which is stored in a separate Mapping Part. For clarity, the contents of the header is shown as a gray box in the figure. This header is followed by multiple Mapping Parts that hold the actual physical addresses of the blocks belonging to that disk file. In this case, the disk file is described by three additional Mapping Parts: the first Mapping Part describes the blocks at offsets 1 through 12; the second Mapping Part describes blocks 15 through 17; and the last Mapping Part describes the last eight blocks at offsets 25 through 32. Notice that there are two (implicit) holes in the disk file, one at offsets 13 through 14 and the other at offsets 18 through 24. We have left out the actual physical addresses, as they are of no importance for the example. Figure 6.10 shows us what a cluster with multiple disk files may look like. In this case the cluster only holds two disk files: disk file (6,1) with 16 blocks and disk file (6,3) with 8 blocks.

6.2.5 Describing Multiple Disk Files using Mapping Parts

The previous sections described how Mapping Parts can be used to describe the mapping information of a single disk file. In LD, the meta-information of each disk file and cluster (including the headers) is described by one or more Mapping Parts. The whole Mapping, describing the meta-information of all disk files and clusters, then, simply consists of the collection of all these Mapping Parts. Figure 6.10 already showed us an example of this. It described one cluster with two disk files in it.

Recall that clients can request LD to physically cluster disk files and disk clusters on disk. If a client requests physical clustering for a particular disk file or cluster, it indicates to LD that such a client is likely to access the blocks of that disk file sequentially, or the client will access multiple disk files of that cluster as a group. Therefore, LD will place the blocks of such disk files close together on disk. However, for quick access to those blocks, it is also important that the mapping information for those blocks are physically clustered on disk. This way LD minimizes the number of disk accesses and the distances of disk head seeks required to access both the mapping information as well as the blocks of the actual disk files on disk.

In short, in order to efficiently find the physical block addresses of the blocks of a disk file, LD must store the mapping information of disk files clustered in the Mapping. LD accomplishes this by storing all Mapping Parts in the Mapping sorted by the logical address field in their MP-Header. The result of this is that all Mapping Parts of one disk file are clustered together and so are all Mapping Parts of disk files that belong to the same disk cluster. Note, that LD stores the mapping information of all disk files and clusters together, irrespective of whether a client has requested physical clustering for that particular disk file or cluster.

6.2.6 Sequence Set

The actual mapping information, that is, the Mapping Parts with the physical addresses and the disk file and cluster headers are stored in the sequence set of the W-tree. Each node of the sequence set is stored on disk in a metadata block, which we call a **Mapping block**. Before we explain the layout of a Mapping block, we first take a closer look at the two parts of a Mapping Part: the **MP-Header** and the **MP-Data**. The MP-Header of a Mapping Part tells what part of which disk file is described by that Mapping Part. The MP-Data part of a Mapping Part holds the actual data of a Mapping Part. Both parts can be stored separately in a Mapping block. Below we discuss each of these parts in a little more detail.

The MP-Header

Table 6.4 shows the fields of the MP-Header, which is a fixed-size structure. The first four fields in the MP-Header form an internal logical block address. It tells us whether this Mapping Part denotes a range of consecutive logical addresses within one disk file or whether it denotes a header. In the former case, the logical address is the starting address of this consecutive range of addresses. The count field denotes the number of logical blocks in this range (i.e., the length of this range). The compression field denotes

Table 6.4: Contents of an MP-Header.

Field	Description
aru	aru_id
cluster	cluster_id
file	diskfile_id
offset	offset
count	the number of logical blocks described by this Mapping Part or the size of header data in bytes
compression	the compression method used
phys_addr	pointer to the MP-Data

which compression method is used to encode the corresponding physical block addresses in the MP-Data part of this Mapping Part. This field can hold the value *Enumeration*, *Sequence*, or *Repetition*. The different compression methods were discussed in detail in Section 6.2.3. The last field in the header is the phys_addr field. The MP-Header and MP-Data part of a Mapping part can be stored separately; the phys_addr field contains a reference to where the MP-Data belonging to this MP-Header can be found on disk. Later on we will see that this MP-Data part resides within the same Mapping block on disk as the MP-Header.

If the Mapping Part holds a disk file or cluster header, the first four fields identify the disk file or cluster whose header is described in this Mapping Part. The count field holds the size of the disk file or cluster header in bytes. The compression field is unused. The phys_addr field points to the MP-Data part, which holds the actual header information.

The MP-Data

The MP-Data part of a Mapping Part holds the actual block information or header information. In the first case, the MP-Data part consists of an array of physical block addresses. In the latter case, the MP-data part contains the contents of the disk file or disk cluster header. In contrast to the MP-Header, the MP-Data part has a variable size.

The Mapping Block

Each Mapping block of the sequence set contains one or more Mapping Parts, each of which holds (part of) the mapping information for a disk file. The Mapping Parts within a Mapping block do not have to describe the same disk file; each Mapping Part can refer to a different disk file. Mapping Parts do not cross Mapping block boundaries. As a consequence of this, there is an upper limit to the size of a Mapping Part. Since a disk file may have multiple Mapping Parts, this restriction does not have any consequences for the maximum size of a disk file. It does, however, put a limit on the maximum size of the header of a disk file or cluster. A header can never be larger than the size of a Mapping block. Since a header is meant to hold only a small amount of information, this restriction should not be a problem. If worthwhile, this restriction could be removed in the future.

Mapping Parts are stored in a Mapping block in such a way that lookups, inserts, and deletes of mapping information in Mapping Parts can be performed efficiently.

A Mapping block contains three parts:

- Mapping-block Header
- Array of fixed-size MP-Header parts
- Array of variable-size MP-Data parts

The Mapping-block Header contains administrative data. The contents of this header is shown in Table 6.5. Each Mapping block is a metadata block and is thus identified by a logical metadata block address, which is stored in the header. The next_block field is a reference to the next Mapping block in the sequence set. The free field stores how many bytes are still free within this Mapping block. The last field nr_entries contains how many Mapping Parts are in this block.

Table 6.5: Contents of a Mapping-block Header.

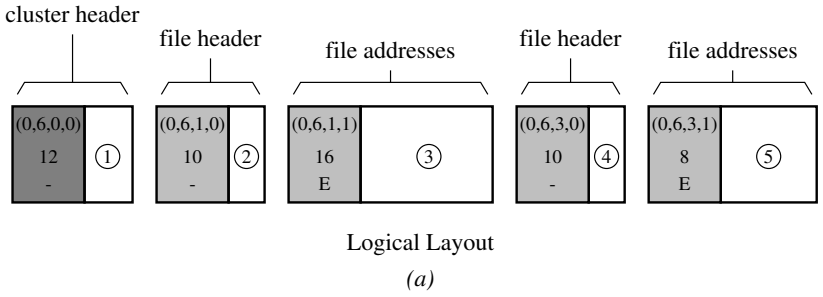
Field	Description
logical_block_address	logical metadata block address of this block
next_block	logical metadata block address of next mapping block in sequence set
free	amount of free space in this mapping block
nr_entries	number of Mapping Parts in this block

The Mapping Parts that are stored in a Mapping block are split in their MP-Header and MP-Data parts. All the MP-Header parts are stored together in an array and so are the MP-Data parts. The reason for storing the header and data parts of a Mapping Part separately in a Mapping block, is that the MP-Headers have a fixed size whereas the Mapping Parts themselves have not. By storing the fixed-size MP-Headers separately in an array, we can accelerate lookups by using binary search. The MP-Headers need to be kept sorted for this reason.

The MP-Data parts are stored at the beginning of the Mapping block and the MP-Headers are stored at the end of the Mapping block, as shown in Figure 6.11(b), which will be explained below. Both parts can grow toward the middle. To implement this scheme, all MP-Headers need an extra field: phys_addr, as was mentioned previously. This field contains a pointer (or index) that points to the start of the corresponding MP-Data somewhere in the beginning of the Mapping block. Note that MP-Headers need to be stored sorted in an array, whereas the corresponding MP-Data parts can be in any order, as long as the pointers in the MP-Headers point to the correct location within the Mapping block.

Figure 6.11(a) shows an example of the logical contents of a Mapping block holding a number of Mapping Parts. This picture shows the configuration of two disk files including their headers in one cluster with its corresponding cluster header. This configuration was

already used before in Figure 6.10. The contents of the MP-Data parts of the different Mapping Parts have been numbered 1 through 5 in the figure, so they can be distinguished from each other. Figure 6.11(b) shows how this mapping information could have been physically stored in a Mapping block. The MP-Headers are stored separately from their MP-Data parts, and the pointer in each MP-Header points to where the corresponding MP-Data part can be found. The MP-Headers are clustered at the end of a Mapping block, and are sorted (in reverse order), so that searching is fast and efficient. The MP-Data parts are in a more or less random order at the front of the Mapping block. In the middle of the block is free space, which can be used to add more Mapping Parts to this block.



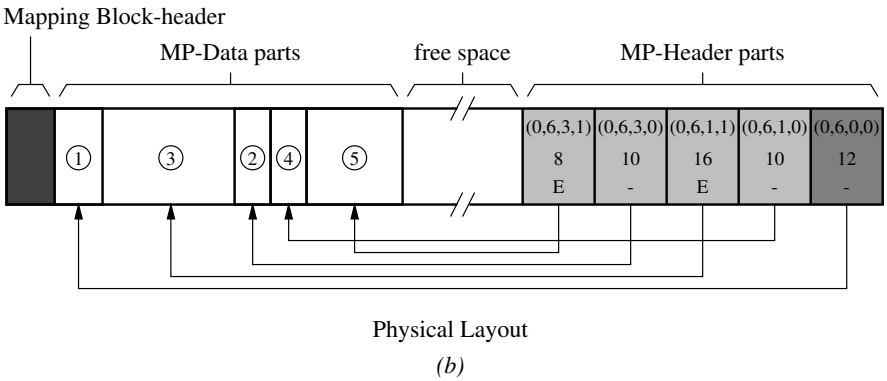


Figure 6.11: The layout of a Mapping block. (a) Logically, this block holds five Mapping Parts which represent two disk files in the same cluster with the corresponding headers. (b) Physically, the Header and Data parts of these Mapping Parts are stored separately in a Mapping block.

There are a number of advantages and disadvantages of physically storing the mapping information in a Mapping block as we described above.

- An advantage is that, as already stated, lookups within a Mapping block can use binary search, which is faster than linear search for large numbers of Mapping Parts in a Mapping block.

- A disadvantage is that the MP-Headers must be kept sorted. Inserting a new Mapping Part, however, can be done relatively quickly. Only the MP-Header of the new Mapping Part needs to be stored in the correct place within the array of MP-Headers, which on average involves shifting half of the already present MP-Headers in the array. The MP-Data part can just be appended to the array of MP-Data parts, if enough free space is available.

- Another disadvantage is that the deletion of a Mapping Part also requires that parts of both the array of MP-Headers and the array of MP-Data parts need to be shifted in the Mapping block, so that the arrays do not contain gaps. An alternative would be to have a defragment process run in the background which manages the fragmented free space in a Mapping block. Although this alternative does not involve less work to be done, it can do so at a later point in time, spreading the work more. Unfortunately, this also complicates the search process because it must deal with holes in the array of MP-Headers correctly.

In short, keeping the Mapping Parts sorted in a Mapping block makes searching a Mapping block fast, at the expense of making the management of Mapping Parts more costly. In Section 6.5 we will introduce a technique which will help to lower the number of inserts and deletes of Mapping Parts in Mapping blocks. This technique, therefore, will make the contents of a Mapping Part more stable, which makes the disadvantages of keeping the Mapping Parts sorted less noticeable.

6.2.7 Index Set

The nodes in the index set of the W-tree are different from the nodes in the sequence set. These index nodes do not hold Mapping Parts, but function only as a search structure on top of the sequence set. Each index node consists of two parts:

- Index-Node Header

- Array of (key, pointer)-pairs

The Index-Node Header contains information about the index node itself. Its fields are described in Table 6.6. The index set potentially consists of multiple layers. The header contains a level field, which denotes on which layer a node is within the tree. The bottom layer of the index set is level 0, and each level higher increases the level by 1. By numbering the levels bottom up, the level of each index node remains stable as the tree increases and decreases in height.

The array of (key, pointer)-pairs are kept sorted so that binary search can be used to speed up the process of traversing the tree. Each node can hold a maximum of about 200 (key, pointer)-pairs, but on average the nodes in a W-tree will, in practice, have a filling degree of around 69% if random inserts dominate [Yao, 1978; Johnson and Shasha, 1989]. We expect that the index set will consist of up to three layers. After all, with three layers and with an average of $200 \times 69\% = 138$ keys in each index node, the index can address over 2.4 million Mapping blocks in the underlying sequence set. Each Mapping Block is 4 KB; therefore, such a sequence set would have a total size of almost

Table 6.6: Contents of an Index-Node Header.

Field	Description
logical_block_address	logical metadata block address of this block
next_block	logical metadata block address of next index block on the same level
level	the level in the Index Set; 0 is bottom level
nr_entries	number of (key, pointer)-pairs in this block

10 GB. Such an amount of mapping information is sufficient for LD to store the block information of most file systems that fit within the size of current hard disks, which can hold up to about 200 GBs worth of data. The actual amount of mapping information that LD needs in order to manage a certain disk depends on how many disk files and clusters are stored on that disk and how effective the use of compression in Mapping Parts is.

6.2.8 Concurrency

The Mapping of LD is used in an environment where multiple threads of execution can be active at any time. To avoid race conditions that can corrupt the tree, accesses to the Mapping must be synchronized. Synchronization can be achieved by using locks on each node in the tree. To increase the degree of concurrency, a distinction between readers and writers can be made. Multiple readers may be granted access to the same node, but a writer must have exclusive access to a node.

Furthermore, LD must also avoid *starvation* and *deadlock* situations. One technique is to define an ordering on all lockable objects (such as nodes within the tree) and require threads to request locks in an increasing order only. This way, deadlocks can never arise. Another technique is the *wound-and-wait* method [Rosenkrantz et al., 1978], which is based on timestamps. This method avoids deadlock situations by aborting ('wounding') a younger thread as soon as an older thread requests a lock that the younger thread currently holds. The younger thread must then start over and try again, but it is allowed to keep its original timestamp. A younger thread that requests a lock on a node currently held by an older thread is allowed to wait until the older thread releases the lock. This way a deadlock (i.e., a cycle of threads, each of which is requesting a lock that the next thread in the cycle holds) cannot occur. This method also avoids starvation since a thread that is repeatedly aborted after requesting locks held by older threads, will eventually become the oldest running thread, at which time it is allowed to obtain all the locks it needs. Therefore, each thread finishes its job eventually.

6.3 FreeMap

The **FreeMap** keeps track of which physical blocks on disk are used and which are not. Conceptually, this data structure is also a table of (key, value)-pairs. This time, however, a key is a physical block address and the value encodes whether the corresponding block

is used or not. Traditionally, such a data structure is implemented by a free block list, which is a list of the addresses of all free blocks on disk, or a bitmap in which each bit represents a disk block, indicating whether it is free or not. For example, the System V file system (S5FS) [Bach, 1986] uses a free block list, and the Berkeley Fast File System (FFS) [McKusick et al., 1984] uses a bitmap to keep track of free blocks on disk.

LD uses a bitmap to implement its FreeMap. The advantage of such an implementation is that bitmaps are very space efficient. Furthermore, a bitmap makes it easier for LD to allocate blocks that improve the clustering of blocks. In order to ensure that the FreeMap is persistent, the FreeMap is stored on disk in logical metadata blocks.

Each bit in the FreeMap represents a physical sector (512 bytes) on disk. Since the number of sectors on a disk is fixed, the size of the FreeMap is also fixed, which is roughly 0.02% of the total size of the disk. Even though the size of the FreeMap is fixed, the blocks of the FreeMap cannot be statically allocated in advance. Fixed locations on disk for the blocks of the FreeMap are not possible due to our requirement of no in-place updates. Since even the FreeMap blocks may not be updated in-place, each updated FreeMap block is written to a new location on disk. As a result the locations of FreeMap blocks are dynamic and cannot be preallocated. We will come back to how the blocks of the FreeMap, and more in general, how metadata blocks are written to disk in Section 6.6.

The FreeMap keeps track of every sector on the disk: blocks holding client data as well as blocks holding LD's own metadata. Consequently, the FreeMap also keeps track of the blocks that store the FreeMap itself. Metadata blocks are 4 KB, so each metadata block is represented by eight consecutive bits in the FreeMap, compared to only a single bit for each client data block.

For efficiency, it is important that the FreeMap is cacheable, which means that access to the FreeMap must have a certain degree of locality, so that caching blocks of the FreeMap in main memory is effective. The FreeMap is updated and consulted whenever blocks are allocated or freed. Since LD tries to cluster blocks on disk, the corresponding bits in the FreeMap that are consulted and updated are also located near each other.

6.4 Logical Metadata Block Addresses

The metadata blocks of the Mapping and FreeMap are addressed via **logical metadata block addresses**. A logical metadata block address is a 4-byte positive number, of which, currently, only the least significant 20 bits are used. Just as the logical block addresses from Chapter 3 have a Mapping that maps them onto physical addresses, so do the logical metadata block addresses. These logical metadata addresses are mapped onto physical addresses with the help of two data structures: the Meta Mapping and the *Root Mapping*. These two data structures are persistently stored on disk and are also referred to as metadata.

The reason for using logical addresses for metadata blocks, instead of directly using physical addresses is that, this way, changes to the physical locations of metadata blocks can be efficiently recorded. Because LD upholds the no in-place update policy for all blocks, LD cannot overwrite metadata blocks in-place. Therefore, whenever clients cause updates to the Mapping or FreeMap, LD is forced to write the changed metadata blocks

to new locations on disk.

As a consequence, the physical addresses of metadata blocks are not stable and cannot be used as stable references. For example, the blocks in the index set of the Mapping contain references to other blocks of the Mapping (i.e., metadata blocks). If these references were actual physical block addresses, then writing a block in the sequence set at another physical location would cause *cascading updates* to all index nodes in the path to the root of the tree that implements the Mapping. However, with logical metadata addresses, the change is limited to an update to the mapping that maps logical metadata block addresses to physical addresses. Unfortunately, the change in this mapping could also result in a cascading update within that mapping. However, we have designed this mapping such that cascading updates are limited. We will return to the topic of cascading updates in Section 6.4.5.

6.4.1 Meta Mapping and Root Mapping

The **Meta Mapping** is a variable-size array of 4-byte physical block addresses, which are the physical addresses of the blocks of the Mapping and FreeMap. The physical block address of a metadata block of the Mapping or FreeMap can be found by using that block's logical metadata block address as an index into the Meta Mapping. The Meta Mapping itself is also stored in metadata blocks on disk. Because of our no in-place update requirement, these Meta Mapping blocks do not have a fixed location on disk; their locations are also dynamic. To keep track of their locations on disk, the Meta Mapping has an index on top of it: the Root Mapping.

The **Root Mapping** is a single metadata block of 4 KB and can contain a maximum of 1024 4-byte physical block addresses: the locations of at most 1024 Meta Mapping blocks on disk. The Root Mapping and the Meta Mapping together form a two-level data structure that is used to map logical metadata block addresses to physical block addresses. The Root Mapping is only a single block and is always kept in-core. How the Root Mapping and Meta Mapping are recovered after a crash is discussed in Chapter 7.

Figure 6.12 illustrates how the Root Mapping, Meta Mapping, Mapping and FreeMap are connected. On the right of the picture are the two main metadata structures: the FreeMap and the Mapping. The FreeMap is an array of metadata blocks each holding the bits that represent whether a physical block on disk is free or not. The metadata blocks of the Mapping form a tree structure, as is depicted in the figure. The blocks of the FreeMap and of the Mapping (including both the index set as well as the sequence set blocks of the Mapping) have logical metadata block addresses.

The dashed arrows between the blocks of the Mapping represent the connections between these blocks within the tree structure: they refer to each other by their logical metadata block addresses. The other arrows in the figure are drawn solid, to indicate that these are references on the basis of physical addresses. For example, the Meta Mapping contains physical addresses of metadata blocks; therefore, a solid arrow is drawn starting at an entry of the Meta Mapping to a metadata block of the FreeMap or Mapping.

At the left of the picture are the Root Mapping and the Meta Mapping. The Root Mapping contains the disk addresses of the blocks of the Meta Mapping, indicated by the solid arrows from the Root Mapping to blocks of the Meta Mapping. The entries in

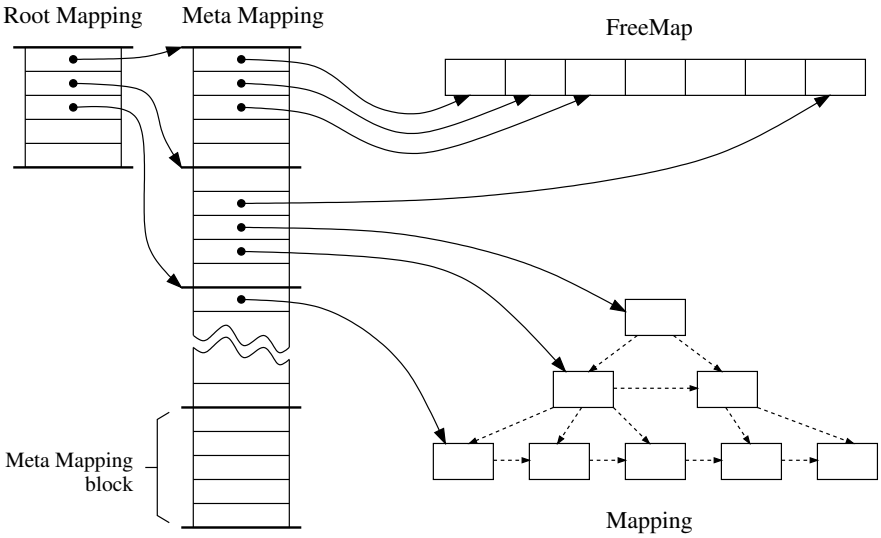


Figure 6.12: Logical connections between the Root Mapping, the Meta Mapping and the metadata blocks of the Mapping and the FreeMap. Entries in the Root Mapping contain the physical addresses of blocks of the Meta Mapping. Entries in the Meta Mapping contain the physical addresses of blocks of the Mapping and the FreeMap.

the Meta Mapping are the physical addresses of the metadata blocks of the FreeMap and Mapping. In short, the physical address of a metadata block of the FreeMap or Mapping can be found by using its logical metadata address as an index in the Meta Mapping. In order to find the correct Meta Mapping block, however, its location must first be found in the Root Mapping.

6.4.2 Using the Root Mapping and Meta Mapping

Now that we have explained the structure and function of the Root Mapping and the Meta Mapping, let us turn to how they are actually used to transform a logical metadata block address into a physical block address. Recall that a logical metadata block address only uses the lower 20 bits of a 4-byte integer. The upper 10 bits of those 20 bits are used as an index in the Root Mapping to find the physical address of the corresponding Meta Mapping block on disk. The lower 10 bits of a logical metadata block address are then used as an index in this Meta Mapping block just found to yield the physical address of the metadata block. A graphic depiction of a logical metadata block address is given in Figure 6.13. Figure 6.14 shows a graphic representation how the physical block address of a metadata block in the metadata area is found. Note that the metadata blocks of the Root Mapping and the Meta Mapping are always addressed via physical disk addresses, and do not have logical metadata block addresses.

Currently, LD only uses 10 bits of a logical metadata address as an index in the Root Mapping; therefore, the Root Mapping can address at most 1024 Meta Mapping blocks.

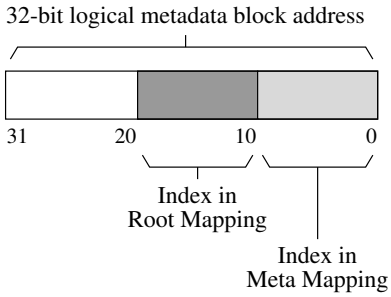


Figure 6.13: A logical metadata block address. Only the lower 20 bits of a logical metadata block address are used. The upper 10 bits of these lower 20 bits are used as an index into the Root Mapping to find the corresponding Meta Mapping block. The lower 10 bits are used as an index into this Meta Mapping block to find the physical address of the metadata block.

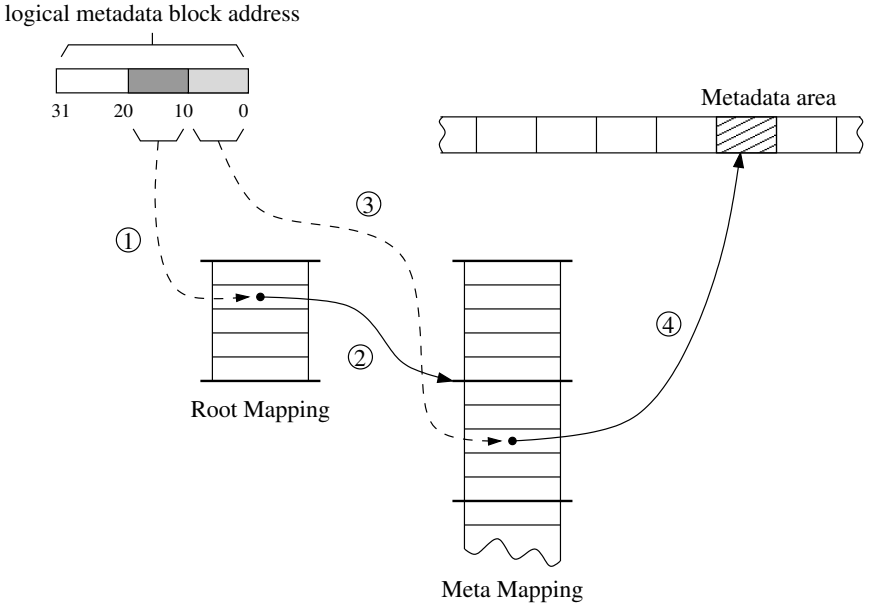


Figure 6.14: Finding the physical block address of a metadata block in the metadata area, given its logical metadata block address. (1) The upper 10 bits of a logical metadata block address are used as an index into the Root Mapping. (2) This entry gives the location of the corresponding Meta Mapping block. (3) The lower 10 bits are used as an index into this Meta Mapping block. (4) This entry yields the physical block address of the desired metadata block. All metadata blocks are always located somewhere in the metadata area.

With each address taking 4 bytes, the Root Mapping is exactly 4 KB, one metadata block, large. This Root Mapping is kept cached in-core at all times. However, for recovery purposes, the Root Mapping is written to disk during a checkpoint (see Chapter 7). Each Meta Mapping block can also hold 1024 block addresses of metadata blocks. Therefore, the Meta Mapping has a maximum size of 1024 blocks, and can address at most $1024 \times 1024 = 1$ million metadata blocks (i.e., 4 GB of metadata). However, 4 GB of metadata is more than enough for single disk systems. In the future, the size of the Root Mapping can be easily increased since its size is not fundamental to the design of LD.

6.4.3 Reasons for the Larger Size of Metadata Blocks

Now that we have explained how metadata blocks are addressed, we can explain why a metadata block (4 KB) is larger than a client block (512 bytes). The reason for choosing such a small size for client blocks is to give clients the finest grain control possible over the disk space used by the client's disk files; a finer grain is not supported by the disk hardware. Thus, it may seem logical to use the same size for metadata blocks as well. An advantage of choosing small metadata blocks is that a cache holding recently used metadata blocks for future reference, can more efficiently cache only those parts of the metadata that are really necessary. As a result, a cache can follow the working set of the metadata more closely, because it can use the available cache memory more efficiently.

However, there are two reasons for choosing a larger size of metadata blocks. The first is that a cache is not only used to hold recently used information, but also to prefetch information which is likely to be used in the near future. Because of the order in which LD stores metadata in the Mapping, sequential access to disk files and locality in accessing disk files of the same cluster results in accesses on metadata that are stored close together on disk. Therefore, since larger metadata blocks can hold the metadata of more clustered disk files or even disk clusters, it is expected that, in case of clustered access, prefetching metadata will be more efficient with larger metadata blocks.

The second reason for larger metadata blocks is to keep the total number of metadata blocks small. A small number of metadata blocks keeps the size of the Meta Mapping small, so that large parts of it can be cached in main memory most of the time. Caching decreases the number of disk accesses necessary to access the Mapping or the FreeMap. The size of metadata blocks is also important for the fan-out of the W-tree used to implement the Mapping. Larger index nodes in the index set of the W-tree can hold more (key, pointer)-pairs, which means that fewer index nodes are necessary. As a result, less main memory is needed to cache most of the index set. In addition, the height of the W-tree is smaller, which results in faster searches.

6.4.4 Special Logical Metadata Block Addresses

Logical metadata block addresses are assigned to new metadata blocks when they are created. Available logical metadata block addresses are found by scanning the Meta Mapping for unused entries. Such entries are recognizable by a `nil` value. However, there are a number of special logical addresses that are pre-allocated to certain metadata blocks. For example, all blocks of the FreeMap have pre-allocated logical metadata block addresses.

Since the size of the FreeMap is fixed and is determined by the size of the disk, the number of metadata blocks needed for the FreeMap is also known in advance and never changes. The logical metadata block addresses for these FreeMap blocks are assigned when the disk is prepared to be used by LD. The preparation of a disk is similar to the process of creating an initial UNIX file system on a disk with a command such as `mkfs`. A consecutive range of logical metadata block addresses is assigned to the blocks of the FreeMap. The size of the FreeMap and the logical metadata block address of the first FreeMap block are stored in the superblock. This way, LD can derive the logical addresses of all the FreeMap blocks when at startup LD reads the superblock.

Another metadata block whose logical address is also stored in the super block, is the metadata block containing the root node of the W-tree of the Mapping. Given the address of the root node, LD can find every block of the Mapping on disk. Without it, LD could not read the Mapping containing the addresses of client blocks.

6.4.5 Cascading Updates

A cascading update in LD is the phenomenon that an update to a single block may result in an update to another block, which, in turn, leads to an update to another block, etc. All these blocks need to be written to disk, which means that the original single update to one block may actually require many blocks to be written; this phenomenon is bad for performance since it lowers the effectively used disk bandwidth. Therefore, this cascading update phenomenon must be kept to a minimum. In this subsection, we discuss where and how the cascading update may occur within LD.

In the beginning of Section 6.4 we already briefly mentioned the issue of a cascading update problem within the Mapping. This problem, however, is only a small part of another, potentially much larger, cascading update within LD. To illustrate this problem, consider the following scenario. Suppose a client writes a single logical block, C_1. This block is assigned a location in the log area and a corresponding entry is made in the Mapping. This assignment results in an update to a FreeMap block and an update to a Mapping block; we name these new metadata blocks F_1 and M_1, respectively. For simplicity, we assume that the update to the Mapping only changes a single Mapping block and does not cause a split in the W-tree, which may results in more Mapping blocks to be updated.

Metadata blocks F_1 and M_1 need to be written at new addresses in the metadata area eventually, and therefore, LD allocates two new locations to these metadata blocks, and updates the Meta Mapping accordingly. In other words, the update of FreeMap block F_1 results in an update to another FreeMap block F_2 and an update to a Meta Mapping block MM_1; the update to Mapping block M_1 results in updates to FreeMap block F_3 and Meta Mapping block MM_2. Both these Meta Mapping blocks must be written to disk, and therefore, this writing results in two more updates to the FreeMap (blocks F_4 and F_5) and in two updates to the Root Mapping. Fortunately, the Root Mapping is in-core and is only written to disk during a checkpoint to a pre-allocated location on disk (see Chapter 7); therefore, updates to the Root Mapping will not result in any other updates. However, the updates to FreeMap blocks F_2, F_3, F_4, and F_5 result in more updates to the Meta Mapping and the FreeMap, etc. From this 'simple' example, it is clear that an update to a single

client data block could lead to an enormous cascading update.

Fortunately, in practice, this cascading update is not likely to occur. The reason that this will not occur is the way LD assigns new locations to metadata blocks in the metadata area. LD allocates physical blocks to metadata blocks such that they are located close together (see Section 6.6). The allocation bits for these addresses in the FreeMap are, therefore, also close together, and it is very likely that they are located within the same FreeMap block (each bit in the FreeMap represents a 512 byte block; therefore, each 4 KB FreeMap block can contain the information for 4096 other 4 KB metadata blocks). Consequently, many of the updates to the FreeMap in our example above are actually to the same FreeMap block, which means that the number of updated metadata blocks is much smaller than the example may suggest at first sight.

With this knowledge about LD's allocation algorithm we can look at our previous example again. The client's update to block C_1 still results in updates to FreeMap block F_1 and Mapping block M_1. The allocation of disk space for metadata blocks F_1 and M_1, however, is such that the corresponding updates to the FreeMap are most likely to result in updates to a *single* FreeMap block (F_2) only. In other words, FreeMap block F_3 in our example above is the same block as FreeMap block F_2. The updates to Meta Mapping blocks MM_1 and MM_2 are likely to be in two different blocks, as the logical metadata block addresses of FreeMap block F_1 and Mapping block M_1 are likely to reside in different Meta Mapping blocks. The allocation of free space to write both Meta Mapping blocks, however, is again likely to result in updates to the same FreeMap block F_2; therefore, FreeMap blocks F_4 and F_5 in our example above refer to the same FreeMap block F_2.

To complete the example, writing FreeMap block F_2 will result in an update to another Meta Mapping block (and subsequently the Root Mapping) and an update to the same FreeMap block F_2. In conclusion, the cascading update is limited to only five other updates to metadata blocks because many updates to the FreeMap are to the same block. LD applies all these FreeMap updates to an in-core cached copy of the FreeMap block, and has to write this block to disk only once. The overhead of five metadata blocks when a client updates a single client data block may seem large. However, in practice, this overhead is amortized over multiple client updates because caching of metadata blocks in LD lowers the average overhead per client update. Caching enables LD to accumulate the effects of multiple client updates to a single cached metadata blocks before LD writes it to disk.

In the next sections we introduce other techniques that help to lower the overhead of writing metadata to disk even further.

6.5 The Differential Technique

In the previous two sections, we have discussed the two main metadata data structures of LD: the Mapping and the FreeMap. Maintaining these two data structures is essential to keep LD running. However, from the viewpoint of a client, updates to LD's metadata are merely overhead. Clients are interested only in their own client data. In this section, we discuss a technique how LD decreases the overhead of updates to the Mapping and

the FreeMap. Fortunately, updating the data structures themselves while they are in main memory is not very time consuming. The main overhead is formed by disk accesses that are needed to read and write metadata blocks to and from disk. Such accesses are relatively time consuming and may use up considerable part of the available disk bandwidth. The amount of disk space that the metadata uses on disk is not much of a problem because the amount of metadata is relatively small and disk space is relatively cheap.

LD achieves the goal of minimizing the overhead of disk accesses to metadata in two ways. First, LD tries to minimize the *number* of times it reads or writes metadata blocks to and from disk. Lowering the number of *disk reads* is done by caching metadata blocks. The number of *disk writes* can be minimized by using a differential technique, which will be discussed below. The second way LD uses to minimize the overhead of accessing metadata is by grouping several metadata block writes together and combining them into larger physical writes. The particular technique that we have developed to accomplish this effect is called the **staccato write**, which will be discussed in Section 6.6.

The overhead of metadata updates is mainly due to the fact that the amount of metadata written is large compared to the actual update that is made to the Mapping or FreeMap. For example, a typical update to the Mapping involves altering only one entry in a Mapping Part. This change may be so small that it only involves changing a few bytes. Unfortunately, this change to the Mapping is made persistent by writing the whole Mapping block containing the Mapping Part to disk. Therefore, 4 KB is written to disk of which most bytes are unchanged. An update to the FreeMap illustrates this effect even better. A single update to the FreeMap effectively only flips one or eight consecutive bits in a metadata block. However, to make this update persistent a 4 KB metadata block of the FreeMap would be written to disk. Actually, due to the cascading update effect, even more 4 KB metadata blocks would be written to disk as writing a FreeMap metadata block would cause an update to the Meta Mapping, etc. In other words, the effective bandwidth that is achieved to write the actual updated bytes to disk is rather low. Luckily, LD can increase the effective bandwidth by delaying the write of a metadata block until it incorporates several changes. The technique used for this is the differential technique.

The **differential technique** used works as follows. Each data structure that holds metadata is split into two parts: a **basic** part and a **differential** part. The basic part is the main data structure and holds the lion's share of the stored metadata. The differential part is small and contains pending *changes* to the basic part, that is, the entries in the differential part contain new data that *override* data in the basic part. The basic idea behind the differential technique is that updates to the metadata are not directly applied to the basic part, but are first stored in the differential part. A separate **merge process** is responsible to apply these updates from the differential part into the basic part at a later time.

In short, a differential part acts as a space-efficient write-cache for updates. Each differential part is simply a list of recently executed updates, which are stored in the list until enough changes can be merged into the basic part efficiently. The differential and basic parts together hold the up-to-date contents of the metadata. The differential technique is more space efficient than normal caching because the differential part contains only updates. With normal caching a single update to a data structure meant caching an entire block of data that was effected by the update.

Each differential part can hold a fixed number of updates; currently, several thousand updates. Since the differential parts hold only the actual updates, that is, changes to the Mapping, FreeMap, or Meta Mapping (which are small), the differential parts themselves are relatively small. Therefore, the differential parts are, in principle, stored only in main memory. However, a copy of it is written to disk during a checkpoint (see Chapter 7) to support recovery. During normal operation, the differential parts are in-core only.

The names of these new data structures are formed by prefixing the word *basic* or *differential* to the previously introduced names of the data structures. The two data structures holding the mapping information are called the **Basic Mapping** and the **Differential Mapping**. We will also refer to these as BM and DM, respectively. The FreeMap is split in the **Basic FreeMap** and the **Differential FreeMap**, or the BFM and DFM, respectively. The basic versions of the Mapping and the FreeMap are as we described previously in Sections 6.1 through 6.3. To minimize the overhead of updating the Meta Mapping, the differential technique is also used for the Meta Mapping. Consequently, there is a **Basic Meta Mapping** or BMM and a **Differential Meta Mapping** or DMM. The BMM is as we have explained in Section 6.4.

The differential parts of the Mapping and the Meta Mapping contain two types of entries. These two types correspond to the two types of update operations that the basic parts of metadata data structures, logically, accept: *store(key, value)* and *delete(key)*. The *store* operation inserts a new (key, value)-pair in the data structure, or modifies an existing entry. For instance, a store operation on the basic part of the Mapping could store a new (logical disk address, physical disk address)-pair to indicate that a logical block has been created or has a new location on disk. The *delete* operation removes a (key, value)-pair from the data structure.

The differential part of the FreeMap only contains one type of entry: *set(key, value)*. This operation corresponds to a set operation on the basic part of the FreeMap which updates the bitmap to indicate a block has been allocated or freed. The FreeMap does not support a delete operation since the entries in the FreeMap are only modified and never deleted.

Note that Section 6.1.3 introduced the interface of the Mapping, which allowed a range of addresses to be stored or deleted. To accommodate this, the entries in the differential parts can also represent store and delete operations for ranges of (key, value)-pairs. However, for simplicity and clarity, in the following examples we will assume that operations in a differential part only concern a single key.

Table 6.7 shows the possible values of the entries in the differential parts of LD's metadata structures. The Differential Meta Mapping accepts entries concerning changes to metadata blocks, which are identified by their logical metadata block addresses. A store operation means that the metadata block has been given a new physical address, and a delete operation means that the metadata block has been deleted. The Differential FreeMap only accepts set operations. Each set operation either records that a physical block is free or used. The Differential Mapping contains store or delete entries concerning logical blocks or header information, which are both identified by an internal logical block address.

Conceptually, new entries for the differential part are always appended to the end. Therefore, the only operation allowed on the differential part is *append(operation)*. The

Table 6.7: Entries in the differential parts.

Metadata data structure	Entry in the differential part
Meta Mapping	operation : *store* or *delete* key : logical metadata block address value : physical block address
FreeMap	operation : *set* key : physical block address value : *free* or *used*
Mapping	operation : *store* or *delete* key : internal logical block address value : physical block address or header

operation is either a store or a delete operation which must be applied to the basic part. However, the differential part does not simply append the operation to the end of a list. In certain cases, a newly appended operation cancels other pending operations that were already in the differential part. For example, suppose a client writes a block of a disk file, and a little later, overwrites the same block again. In this case, the first write operation appends an entry for a store operation in the DM. However, the second write also generates a store operation which is also appended to the DM. This second store operation cancels the first store operation, and therefore, the first store operation can be removed from the DM. In short, the differential part uses its knowledge of the information stored in its entries (i.e., operations) to store as little entries as possible; newer entries can cancel older entries. By storing less operations in the differential part, less updates have to be made to the basic part by the merge process. These optimizations are similar to the ones we have introduced in Section 5.6 for log-tuples in the in-core segment.

6.5.1 Advantages of the Differential Technique

The advantage of this scheme is twofold. First, the use of the differential technique increases the chance of being able to incorporate several updates to a single metadata block of the basic part, which leads to a better disk bandwidth utilization when LD must write a metadata block to disk. Indeed, when the differential part holds many updates, chances are that it holds two or more updates that, when applied to the basic part, would update the same metadata block of that basic part. This situation becomes more likely if the system has a relatively small *working set* or even a small *hot spot*, which means that not all blocks have the same chance of being updated. Instead, within a certain time frame, a set of relatively few blocks are updated more frequently than others.

For LD, a hot spot results in multiple updates to only a few metadata blocks. Therefore, when the merge process selects such updates and moves them from the differential part to the basic part, only a few metadata blocks of the basic part would be updated. Since each of these block contains the result of multiple updates from the differential part, writing these blocks to disk yields an effective bandwidth utilization that is higher than when

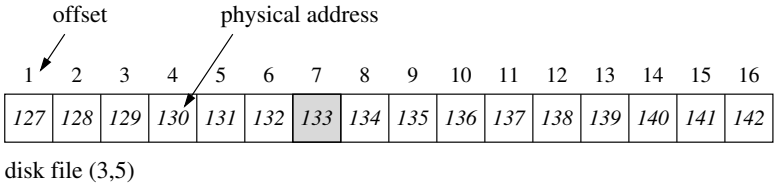
each of these blocks would have been written multiple times; once for each of the updates accumulated in the differential part.

The second advantage is that sometimes no metadata blocks of the basic part have to be written to disk at all. This situation occurs when a new update to a metadata data structure cancels the effect of a previous update. Such updates are likely to occur in LD since LD usually writes updated blocks temporarily in the log, and moves them back to the storage area at a later time. If an updated block belongs to a group of blocks that must be stored physically clustered on disk, the cleaner process may frequently decide to move the updated block back to its original location in the storage area, in order to restore the clustering property. In other words, such a block undergoes two location changes, which are updates to the Mapping: first, when writing it in the log; second, when writing it back to its original location. The net effect of these two changes is that the block ends up on its original location, and therefore, not only the first update in the differential part can be cancelled, but also the second one.

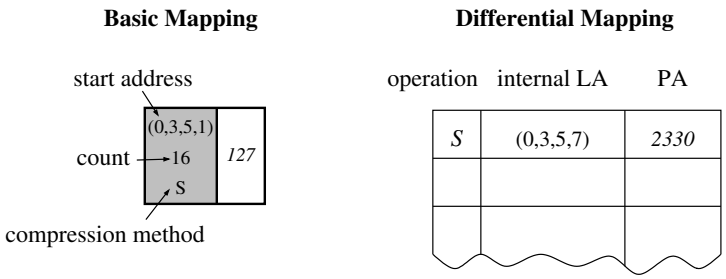
To illustrate the above, consider, as an example, the situation where a client modifies one block of an already existing disk file. Suppose the disk file (3,5) contains 16 blocks, and these blocks are physically perfectly clustered on disk in the storage area, starting at physical address *127*. The disk file is depicted in Figure 6.15(a). The mapping information for this disk file is stored on disk in the BM, the basic part of the Mapping. The Mapping Part in the BM representing this information is shown in Figure 6.15(b). Now, the client modifies the seventh block of this disk file, which has logical address (3,5,7). The new version of that block is written into the log, and is assigned new physical address *2330*. A store operation to change the (key, value)-pair in the Mapping for that block is generated: *S*((3,5,7), *2330*). The *S* identifies this operation as a store operation. This entry is appended to the DM, the differential part of the Mapping, which is depicted in Figure 6.15(c). After a while, the cleaner process tries to move data blocks from the log back into the storage area. It finds the single block of our disk file in the log, which breaks the clustering property of the disk file. In order to fix the clustering of the disk file, the best and simplest way is to copy the new block back to its old position (*133*) next to the other blocks of the same disk file. The cleaner process can reach this conclusion by looking at the physical layout of blocks of disk file (3,5) in the BM. There it can discover that the updated block (3,5,7) had a previous location which is physically clustered with the other blocks of disk file (3,5). If the previous location of (3,5,7) is still available (which is likely to be the case), the cleaner process will move block (3,5,7) back to its original location in the storage area. After the copy, the cleaner process can remove the entry *S*((3,5,7), *2330*) from the DM since the update has now been canceled. Notice, that in this scenario, the BM has not changed, and as a result no blocks of the BM have to be written to disk. All changes have been done to the DM, which is an in-core data structure.

6.5.2 Disadvantages of the Differential Technique

The disadvantage of the differential technique is that the information, which is normally in one data structure, is spread over two data structures. As a result, lookups of entries are more complicated. Instead of consulting only one data structure, lookups must first consult the differential part to see if it holds entries that affect the requested information.



(a)



(b) *(c)*

Figure 6.15: The effects of the Differential Mapping on cleaners. (a) Disk file (3,5) contains 16 consecutive blocks. The seventh block is shown gray to indicate that it is being updated by a client. (b) The Mapping Part with the mapping information for the blocks of disk file (3,5), which is stored in the Basic Mapping on disk. (c) After updating block (3,5,7), an update to the Mapping is generated, which is stored in the Differential Mapping. The entry in the Differential Mapping is a *store* operation (*S*), denoting that the contents of the logical block with internal logical block address (LA) (0,3,5,7) have been stored at physical block address (PA) *2330*.

If not, the search continues in the basic part.

Note that LD does not need to actually apply the operations in the differential part on the basic part in order to be able to see what the effect of these operations will be. The effects of operations in the differential part on the basic part are clear from simply looking at the entries themselves. Each entry in the differential part is either a *set*, *store*, or a *delete* operation on the corresponding basic part. Each entry indicates on which key (logical block address in the Mapping, logical metadata block address in the Meta Mapping, or physical block address in the FreeMap) it operates. Therefore, during a lookup, LD can simply see whether an entry in the differential part affects the requested information of the lookup. If so, a set or store entry means that the requested (key, value)-pair has been updated, and its new value is the one stored within the store entry. A delete entry means that the corresponding key is not present in the data structure (i.e., Mapping or Meta Mapping) anymore. In both cases, the requested information has been found; the basic part does not have to be consulted anymore.

For example, if LD needs to find the physical address of a particular logical block, it first looks in the DM. If that requested block has recently been rewritten or deleted, a corresponding update entry will be present in the DM. If the requested block has been rewritten, a store entry is present in the DM that gives the new physical address for that block. If the requested block has been deleted, a delete entry for that block will be present in the DM. If the DM does not contain information about that block, the search for the physical address of that block continues in the BM. If the block is not present in the BM as well or it contains the value NIL, the block does not exist, otherwise the BM holds the requested physical block address of that logical block.

6.6 The Staccato Write

The other way to minimize the overhead of updating metadata is by making the process of writing metadata blocks to disk more efficient. The metadata blocks are written into the metadata area on disk. This area lies somewhere in the middle of the disk, which reduces the size of the average seek that is required when accessing the metadata area. The size of this area varies with the amount of metadata that LD uses. We will return to this topic in Chapter 8.

The metadata blocks in LD are the blocks from the Basic Mapping, the Basic FreeMap, and the Basic Meta Mapping. The Root Mapping and the differential parts of the previously mentioned metadata data structures are written into the checkpoint area, but only when making a checkpoint. The checkpoint area is treated in Chapter 7.

6.6.1 Introducing the Staccato Write

In Chapter 5 we presented the method of collective writes to write data blocks to disk efficiently. We could use this method again to write metadata blocks into the metadata area. However, in this particular case, we can use an even more efficient method, which we have named the **staccato write** method, or simple the staccato method. Although this method is somewhat similar to collective writes, which LD uses to efficiently write client data blocks to disk, these two methods do differ on a few significant points, which we will

discuss later on. Similar to collective writes, the staccato method never updates metadata blocks in the metadata area in-place, and does not write metadata blocks individually, but as a group which leads to a better utilization of the disk bandwidth. The metadata area is used as a cyclical buffer of blocks. New or updated metadata blocks are first accumulated in main memory and then written to disk together.

Where the staccato method differs from collective writes, is how the blocks are actually written to disk. The staccato method uses a variant of the first-fit algorithm to find free space to write individual blocks into the metadata area. Whenever LD needs to write a metadata block, it writes the block in the first available free spot it finds while scanning forwards in the metadata area starting from the position where it wrote the previous metadata block. When the scan reaches the end of the metadata area the scan continues at the beginning of the area again. When the free space in the metadata area is fragmented, this write process results in a disk head that skips over the used blocks in the area while it writes small amounts of metadata blocks in the free spaces in between. The staccato write method gets its name from the image of a disk head turning on and off while it scans the surface of the disk sequentially. This write method is in contrast to the collective writes method, which always writes a group of blocks to disk in a single consecutive write operation, which, therefore, requires a contiguous range of free space on disk large enough to hold the group of blocks.

By carefully maintaining the size of the metadata area we can guarantee an upper limit to the time it takes to write x blocks using the staccato method in the metadata area. For example, by keeping the metadata area at least twice the size LD needs for its metadata, then, in the worst case, only 50% of the disk blocks in the area are available for a staccato write. Depending on the fragmentation of the free space in the metadata area, LD can select a favorable position within the metadata area to start the staccato write. For example, LD could start at the beginning of the smallest range of consecutive disk blocks within the metadata area that contains enough free blocks to hold the amount of metadata that is to be written. In order to do this, LD should first scan the FreeMap to find such a range of blocks. The result of this optimization is that writing x blocks with the staccato method costs at most twice as much time as a sequential write of x blocks, which is the most efficient way to write x blocks.

One possible worst case situation is when only every other block in the metadata area is free. In that case, writing x blocks with a staccato write takes twice as long as a sequential write of x blocks. It takes longer because with a staccato write, the disk head has to cover twice the distance needed to write x blocks sequentially since it has to skip over the used blocks. If the free space is less uniformly fragmented, the staccato write incurs a smaller penalty, because LD can start the staccato write in a range that is less than 50% used.

To summarize the similarities and differences between collective writes for log data and the staccato method for metadata, we can say that both methods first accumulate dirty blocks in main memory in order to write them to disk in a group. However, a collective write writes its blocks as a physically consecutive segment to disk, and therefore, needs a contiguous range of free space on disk. In contrast, a staccato write simply skips over the used blocks. As a consequence, the collective write method needs a cleaner process to create the necessary contiguous free space on disk to hold new segments while the

staccato method does not. This cleaner process could be referred to as a *defragment process*. Our claim is that, overall, the staccato write method is actually more efficient than the combination of the collective write method including the required defragment process. In the next section we will support this claim by giving a simple calculation.

6.6.2 Comparing Collective Writes and Staccato Writes

In this section, we will look at how much it costs to write one segment's worth of data to disk when using the collective write method and when using the staccato method. With the collective write method, a cleaner process must first make room in the metadata area to hold the new data. The cleaner creates this room by reading in a number of old segments and cleaning them. *Cleaning* consists of identifying the *live* blocks in the old segments and writing them in a new segment. Subsequently, the old segments are marked as free and can be reused to hold segments. With 'live' blocks, we refer to blocks that are still valid, as opposed to blocks that have become obsolete because they have either been deleted or newer versions of them have been written elsewhere.

Let us suppose that on average 10% of the blocks in an old segment are still alive. This supposition means that the cleaner has to read in 10 segments to collect one segment's worth of live data. This new segment, which contains the live data, is written to disk which allows the original 10 segments to be returned to the pool of free segments again. Note that since a new segment is written during this process, in effect only 9 empty segments have been created. Therefore, in order to create 9 empty segments, the cleaner has to read 10 segments and write 1 segment. The result is that per empty segment created $\frac{10+1}{9} = \frac{11}{9}$ segments have been read or written. For each segment that is written with the collective write method, these cleaning-overhead costs must also be paid, so in order to write one segment, $1 + \frac{11}{9}$ segment's worth of data must actually be read or written (the sum of writing the segment itself and the cleaning costs that were necessary to create a free segment).

The above situation is illustrated in Figure 6.16. In Figure 6.16(a) a small metadata area is shown. It has room for eleven segments. Only one segment slot is completely free. The other segments are used and contain some live data. The cleaner reads in these used segments, collects all live data from them and writes a new segment with the live data into the last empty segment slot. After this cleaning process, the situation is as depicted in Figure 6.16(b). Figure 6.16(c) shows how a segment with new data is written in the first of the newly created free slots in one large sequential write.

Now let us turn to the staccato method. This time, there are no cleaning costs to create free space. However, we now have to skip over live blocks while writing to disk, which takes time. If we assume that skipping over blocks takes the same amount of time as reading those blocks, we can make the following calculations. Again, suppose that on average 10% of the blocks in our metadata area are alive. If we want to write one segment's worth of data to that metadata area, we cannot write them in one large sequential write, since on average 10% of the blocks in the segment would then overwrite live blocks. Therefore, we have to skip over these live blocks (which, we assume, costs the same as reading these blocks), which means that, as we write the entire segment's worth of data, the disk head covers an extra 10% more disk space than the original size of

Figure 6.16: Cleaning the metadata area and writing a new segment in the metadata area with the collective write method. (a) The metadata is heavily fragmented. The cleaner creates a contiguous range of free space by collecting 'live data' together. (b) The result of creating free segments. (c) A new segment is written into the first free segment with a collective write.

the segment. However, the extra space that is used to hold the extra 10% of data, is also filled for 10% with live data blocks. So, writing the last 10% of the segment also has to skip over live data blocks, which means writing the last 10% costs an extra 1% more. To write the extra 1% also has a 0.1% extra overhead, etc. This geometrical series converges to 1.11111111... = $\frac{10}{9}$. Therefore, on average to write one segment's worth of data costs the equivalent of writing $\frac{10}{9}$ segment's worth of data. The staccato method is shown in Figure 6.17. It shows what happens if data blocks are written into the metadata area which still contains live data blocks. The new data blocks that are to be written into the metadata area are shown at the top of the figure. However, when writing these data blocks in the metadata area, the write process has to jump over the spots containing live data blocks. These live data blocks are represented by the darker patches in the disk at the bottom of the figure. The result of not overwriting live data blocks is that the write has to scan over a larger amount of disk space than strictly was necessary to hold the new data blocks, to complete writing these data blocks to disk. The extra space required is on average $\frac{1}{9}$-th of the actual size to be written, when the metadata area is only filled for 10%.

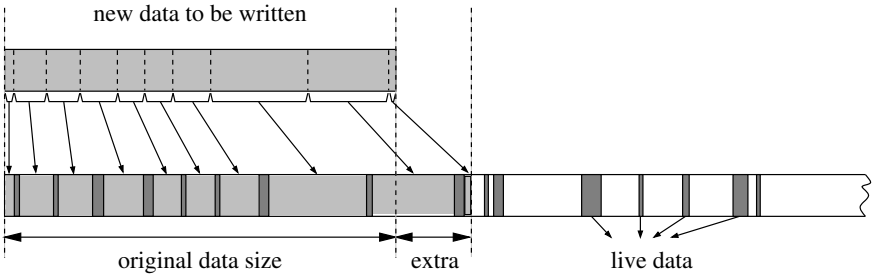


Figure 6.17: Writing data in the metadata area using the *staccato* method. The amount of data to be written (depicted at the top) is actually written in smaller amounts, each of which is written spread over the disk (shown at the bottom). The darker patches represent live data, which must be skipped over by the staccato method.

Comparing the two methods, it is obvious that, at least in our example, the staccato write is a factor of 2 better ($1 + \frac{11}{9} = \frac{20}{9}$ versus $\frac{10}{9}$). However, it can easily be shown that the percentage of live blocks within the old segments has no influence on the overall outcome; the staccato method always reads and writes a factor of 2 less data. In other words, the staccato method requires only half the work of the collective write method. For completeness we give the formulas for the costs for writing one segment's worth of data when using the collective write method and the staccato method:

$$C_{\text{collectivewrite}} = 1 + \frac{\frac{1}{f_{\text{live}}} + 1}{\frac{1}{f_{\text{live}}} - 1} = \frac{2}{1 - f_{\text{live}}}$$

$$C_{\text{staccato}} = \sum_{n=0}^{\infty} (f_{\text{live}})^n = \frac{1}{1 - f_{\text{live}}}$$

where

f_{live} = the fraction of live blocks per segment

Both methods can be improved. For example, the cleaner process can run during periods when the disk is idle. This improvement means that the cleaning overhead will not always cause a hiccup in the response time for clients, which makes the overhead less noticeable, although the amount of work to be done remains the same. The cleaner can pick segments to clean that have a low percentage of live blocks to minimize the overhead of cleaning. However, notice that we can also improve the performance of the staccato write method if we add a cleaner that only runs during idle periods. If there is enough idle time available for a cleaner to defragment the metadata area, both the collective write method and the staccato method including a cleaner perform equally well because there is always enough free space to write a new segment in one large disk write. However, the staccato method performs better than the collective write method when the cleaner cannot complete its work during idle time. Another example of how to improve the staccato method has already been mentioned above: select a starting point of the staccato write that will result in the least amount of skipping. All in all, the staccato write does seem to perform better and is less complex.

However, if the staccato write is better than the combination of collective writes and cleaning, why do we not use the staccato write for writing log segments with client data in the log area as well? The reason is that the staccato method also has its drawbacks. One important disadvantage is that over time the data blocks will be scattered all over the area used. This disadvantage has lead to one of the two reasons why we have decided not to use the staccato method to write log segments to the log area on disk. The first reason is that log segments are used to recover the disk after a crash. LD must know the order in which the log blocks are written, and which blocks belong to the same log segment. With the staccato method the blocks would be scattered, which would make recovery more complex. The second reason is that LD already needs cleaner and reorganizer processes to try to cluster data blocks according to the client's wishes for efficient sequential reading. Therefore, part of the overhead of cleaning log segments is already present.

On the other hand, metadata blocks do not suffer much from being scattered over the metadata area. Metadata blocks are not likely to be read sequentially in large amounts. The fact that the information in metadata blocks, such as mapping information, is already clustered within each block, often is enough to exploit the locality that access patterns exhibit on metadata. Therefore, at that time, we decided that adding an extra reorganizer process to cluster the data in the metadata area did not have a high priority (see also our discussion on the size of a metadata block in Section 6.4.3). Later, however, some experiments with a prototype implementation of LD (see Chapter 9) showed that metadata clustering could improve LD's performance significantly, and therefore, support for metadata clustering has later been added to LD's design (see Section 9.7).

6.7 Keeping Metadata Consistent with Client Data

Until now we have not mentioned when LD writes metadata blocks to disk. This topic is especially important since data integrity is one of our main goals of LD, and LD's

metadata holds information concerning the locations of LD's client data blocks. The consistency of client data blocks are guaranteed by the log, which contains information to redo any changes when necessary after a crash. However, LD's metadata blocks are not written with log tuples. Therefore, after a crash, newly written metadata blocks are not recoverable.

Fortunately, these metadata blocks do not have to be recoverable. Recall that a checkpoint is a snapshot of a consistent state of the disk, which can be restored at a later point in time. This checkpoint is kept on disk and is not modified until after a following checkpoint has been successfully made. Consequently, all updates to LD's metadata that were made up to the point when a checkpoint was made are recoverable after a crash because they are incorporated within the checkpoint. On the other hand, after a crash, all updates to LD's metadata that were done *after* the last successful checkpoint, are lost. However, client updates that triggered the changes to the metadata in the first place, are recoverable because they are logged in log tuples on disk. Therefore, restoring the latest checkpoint and replaying these logged client updates will result in the same (or equivalent) updates to the metadata. In other words, any updates to LD's metadata between two successful checkpoints do not have to be recoverable from disk because they can be redone as a result from replaying the log.

As a result of this, the moment when LD's metadata blocks are written to disk is irrelevant as long as they have been written at a checkpoint. Furthermore, the order in which metadata blocks are written to disk, is also irrelevant. In short, LD can write updated metadata blocks to disk whenever it wants to and in any order. One advantage of this property is that the merge process, which integrates pending changes from the differential part into the basic part, can do its job without having to worry about recovery. However, for internal consistency, it must make sure that moving an entry from the differential part into the basic part is executed as an atomic action. Other processes may not see any temporary in-between state because that would be inconsistent.

The only moment when the state of client data and metadata are guaranteed mutually consistent on disk, is at the time LD makes a checkpoint. A checkpoint forces LD to create a consistent state of both client and metadata on disk. This is necessary because a checkpoint is the starting point of recovery.

A consequence of using a checkpoint as the starting point for recovery is that LD cannot reuse the disk space occupied by metadata blocks that are part of the latest checkpoint. When LD updates metadata blocks, it cannot mark the old metadata blocks as free to be reused. Reusing that space for other data would mean that the checkpoint is not complete anymore, and consequently, LD cannot recover anymore. Only after a following checkpoint has been successfully made, can these old metadata blocks be marked free and reused, because the new checkpoint contains the new versions of these metadata blocks. The new checkpoint with the new versions of the metadata blocks then function as the starting point for recovery. This topic will be discussed further in Chapter 7.

Chapter 7

The Checkpoint Area

The subject of this chapter is the crash recovery process in LD. The technique used for recovery by LD is *logging* in combination with *checkpointing*. This technique completely differs from the approach used in the earlier design of LD (see Section 1.4.2). The log, which keeps a history of executed update commands from clients, has been discussed in Chapter 5. Therefore, the main part of this chapter focuses on the checkpoint.

The checkpoint area is the place on disk that LD uses to store checkpoints, which are snapshots of a consistent state. Checkpoints play an important part in the process of recovering the state of the disk after a crash. Chapter 4 mentioned the two main purposes of checkpoints. First, checkpoints are necessary to prune the log periodically. Second, checkpoints speed up the recovery process, so that LD can quickly resume its job of serving client commands after a system failure.

This chapter is structured as follows. We start in Section 7.1 by giving a brief introduction of crash recovery in general. Section 7.2 presents how LD uses the technique of using a log and checkpoints to recover to a consistent state after a crash. This discussion is followed by three sections that focus on the use of checkpoints in LD. Section 7.3 discusses the requirements that LD puts on a checkpoint. Next, the structure of a checkpoint is discussed in Sections 7.4 and 7.5. In Section 7.6, we present a general overview of the steps that are taken during the recovery process itself. Section 7.7 shows why LD's recovery process, indeed, restores a consistent state on disk. This chapter ends with two sections in which we look at a number of technical issues. In Section 7.8 we discuss the steps that are taken when LD makes a checkpoint. Finally, section 7.9, the last section in this chapter, shows how LD keeps a checkpoint intact during normal operation.

7.1 Introduction to Recovery

Recovery in general covers a broad range of topics. For example, Gray and Reuter [1993] give the following definition:

> "**Recovery:** The process of masking a fault. In transaction processing systems, [recovery consists of] the mechanism to abort transactions while the

> system is operating, the mechanism to restart the system and recompute the most recent committed state after a system outage, and the mechanism to recover objects from archive copies should the online version of the object be lost."

In this dissertation, however, we use the term recovery in a more restricted sense. We refer to recovery in LD as the process that brings the state of the disk after a *system failure* to a consistent state prior to that failure. This state should have the following two properties:

(1) The state must be *consistent*.

(2) The state must be *recent*.

The first property states that the system may not contain errors (i.e., inconsistencies) after it has recovered, which allows the system to continue its work as normal. The second property implies two things. First, the system must recover to a state in which the system has been before. Second, the time at which the system was in that recovered state should be close to the time at which the system failure occured. This second property guarantees that not too much work is lost. Note that the recovered state does not have to be the *most recent* consistent state prior to the crash, just a *recent* one. This limitation is because it is not always possible to recover the most recent consistent state, especially since many systems use some kind of write caching in volatile memory (RAM) for performance reasons.

For example, we assume that LD stores its in-core segment in volatile memory, whose contents are lost after a system crash. Therefore, the results of an ARU that has committed is still lost if its commitaru log tuple is still in the in-core segment when the system failure occurred. One way to recover the last consistent state is to store the in-core segment in NVRAM. In this dissertation, however, we will discuss the more common situation where only volatile memory is available.

In Section 2.3 on page 28, we mentioned that there are two types of failures: system failures and media failures. The former refers to system crashes due to power failures or software errors. The latter refers to crashes that result from failing media (e.g., a disk head crash, or a disk on which bad sectors start turning up). LD guarantees data integrity only with respect to system failures. Furthermore, we assume that LD behaves with *fail-stop* semantics [Schlichting and Schneider, 1983; Schneider, 1984; Chandra and Chen, 1998]. A system with fail-stop semantics that starts to malfunction stops before it writes any erroneous data to disk. Since we focus only on system failures, this chapter only discusses crash recovery. The terms crash and system failure will therefore be used interchangeably.

LD does not support a more general fault tolerance mechanism. For example, LD has no special support to survive media failures, nor does LD prevent malfunctioning clients from ruining the contents of the disk. If protection against such failures is desired, clients must use other techniques such as making backups or using versioning, respectively. LD is a disk system that simply provides some functionality to recover from system failures in a more or less predictable way.

7.2 Recovery in LD

After a crash, LD must be able to recover to a state in which the integrity of its data is guaranteed. Guaranteed data integrity with respect to system failures is one of the main goals of LD. This goal has lead to the introduction of streams and ARUs in the interface of LD. Since LD defines clear data integrity guarantees for streams and ARUs with respect to recovery after a system failure, these abstractions enable clients to indicate to LD which states they consider to be consistent. With streams, clients can control the order in which their commands are committed, and thus, clients have influence on the state that is recovered after a crash. With an ARU, clients can group multiple commands into a larger unit, which guarantees that after recovery all or none of the commands will have been recovered.

In this section, we look closer at what guarantees are provided by LD's recovery process and at the technique that is used to provide them. Before we continue our discussion of recovery in LD, however, we start by introducing some definitions.

7.2.1 Definitions

LD serializes the execution order of client commands according to the semantics defined on streams. Conceptually, LD executes these commands one at a time, and each command is executed atomically. An actual multithreaded LD implementation, however, may increase its performance by executing commands concurrently, but the end result is still as if the commands were executed one at a time. In short, LD serializes the client commands as it executes them, and therefore, the executed client commands form a (serialized) sequence.

The **state of LD** refers to the data that LD has stored on disk. This state consists of both client data and metadata. The client data of this state forms the **client data state**. Similarly, the metadata of this state forms the **metadata state**. The metadata state contains all of LD's data structures, such as the Mapping and FreeMap. For simplicity, we consider all information stored in LD's Mapping to be metadata, even though the Mapping also holds some client data in the form of headers.

The client data state and metadata state refer to the data that LD stores in blocks on disk. In the remainder of this chapter, however, we will use the term 'client data state' to refer to the data stored in client data blocks on disk as well as to data in any unwritten, but changed, client data blocks that have been cached in main memory. Similarly, the term 'metadata state' refers to the data stored in metadata blocks on disk and cached metadata blocks as well as to data stored in LD's in-core only data structures. Of course, in order for LD to make these data persistent, LD needs to flush the data stored in volatile main memory to disk first.

Thus far, we have only referred to a 'consistent' state. Here, we define what a 'consistent' state is. We define four levels of consistency for the state of LD. Below we give the definitions of each of these levels, followed by a more detailed discussion.

- **Client-data consistency** — a state is said to be *client-data consistent*, if its client data state exactly reflects the client data block changes made by some *prefix* of the

sequence of client commands that have been executed by LD. Note that this definition only considers the changes made to client data blocks, and not the changes made to metadata blocks.

- **Metadata consistency** — a state is said to be *metadata consistent*, if the data structures of its metadata state exactly reflect the changes made to them by some *prefix* of the sequence of client commands that have been executed by LD.

- **Overall consistency** — a state is said to be *overall consistent*, if the blocks in this state exactly reflect the changes made to client data and metadata blocks by some *prefix* of the sequence of client commands that have been executed by LD. Note that overall consistency implies both client data and metadata consistency, but the reverse is not necessarily true. The reverse is not true because a state can be both client-data consistent and metadata consistent, but with respect to different prefixes of the sequence of executed client commands. The prefixes may differ in length, and if so, the state is not overall consistent. In an overall-consistent state, the metadata state and the client data state are said to 'belong together'.

- **Recovery consistency** — a state is said to be *recovery consistent*, if the blocks in this state exactly reflect the changes made to client data and metadata blocks by some *prefix* of the sequence of *committed* client commands that have been executed by LD. Note that in the prefix we only consider the commands that are part of committed ARUs (including single ARUs).

Section 7.1, which discussed recovery in general, stated that after a crash, recovery should bring the system back into a consistent and recent state. With the help of our definitions given above, we can now precisely formulate to what state LD should recover.

> After a crash, LD guarantees that it will recover to a state that is *recovery consistent* and *recent*. Recovery consistency guarantees that the recovered state exactly reflects some prefix of the sequence of executed committed client commands. Recentness means that the prefix should be as long as possible. The prefix starts from the last time the disk was initialized to be used by LD.

Below we will briefly discuss each level of consistency in turn.

Client Data Consistency

Due to the way LD writes its client data blocks to disk during normal operation, LD ensures that, at any moment in time, there is a client data state on disk that is client-data consistent. Such a state exists on disk because, as part of LD's guarantee to execute each client command atomically, LD always writes client data blocks to free spaces on disk. LD never overwrites client data blocks that are still in use by clients, which is reflected, for example, in LD's no in-place update policy. Therefore, there is always a client data state on disk that exactly reflects the changes of a prefix of the sequence of executed client commands. Note that this client data state may contain the uncommitted results of commands that are part of uncommitted ARUs.

After a crash, however, LD can only recover this client data state, if LD can also recover the corresponding metadata that tells LD the exact locations of the client data blocks forming this client data state. Without the metadata, LD cannot determine which of the blocks on disk are part of the client data state that is client-data consistent. Unfortunately, the metadata that LD stores in its metadata block in the metadata area are not immediately recoverable after a crash, as we will see next.

Metadata Consistency

One implication of metadata consistency of a state is that all data structures (i.e., Mapping, FreeMap, Meta Mapping and Root Mapping) of the metadata state are *internally consistent* and *mutually consistent*. Internal consistency of the data structures refers to the integrity of the information stored within each data structure. Mutual consistency refers to the integrity of the connections between the data structures. For example, the Meta Mapping contains references to the blocks of the Mapping, which must exist. Another example is that the blocks occupied by the Meta Mapping and Mapping must be correctly registered as 'used' in the FreeMap.

Unfortunately, recall from Chapter 6 that LD writes its metadata blocks into the metadata area on disk asynchronously. Consequently, LD does not continuously keep the metadata state stored in the latest versions of these metadata blocks metadata consistent. In other words, after a crash, it is not guaranteed that the latest metadata state stored in the metadata area is metadata consistent. In order for LD to recover a metadata state that is metadata consistent, LD needs a method to recover its metadata, as we will show in the remainder of this chapter.

Overall Consistency

A client data state and a metadata state are related to each other: the metadata state contains data structures that hold information concerning client data blocks. For example, the Mapping maps logical block addresses of client data blocks to physical block addresses, and the FreeMap records that the physical disk blocks holding those client data blocks are in use. We define that a client data state and a metadata state 'belong together' if they both exactly reflect the changes made by the same prefix of a sequence of executed client commands. In an overall-consistent state that is the result after executing prefix p of the sequence of client commands, the metadata state has the following properties (in addition to the properties that belong to a metadata-consistent state):

- The Mapping in the metadata state correctly refers to all client data blocks that are in use by clients after executing prefix p of the sequence of client commands, and does not contain any references to other blocks.

- The FreeMap in the metadata state records a disk block as 'used' if and only if the disk block either contains a metadata block or a client data block, used by one of LD's metadata data structures or used by a client, respectively, after executing prefix p of the sequence of client commands.

Note that an overall-consistent state can also reflect the changes of uncommitted ARUs, that is, it can contain uncommitted data. In that case, the meta-information of these uncommitted data blocks are correctly recorded in the Mapping and FreeMap.

The relationship between client data states and metadata states is a many-to-many relationship. For each client data state there are multiple different metadata states that, in theory, belong to it, and each metadata state can belong to multiple client data states. We call two metadata states *equivalent* if they belong to the same client data state. Two equivalent metadata states can differ in the way the information concerning client data blocks is stored within LD's metadata blocks. For example, the information stored in the Mapping can be distributed differently over the sequence set blocks of the W-tree that implements the Mapping. Another example is that the metadata blocks may use different logical metadata block addresses, which results in differences in the Meta Mapping and Root Mapping. To clients, however, these two equivalent metadata states are indistinguishable. Two client data states that belong to the same metadata state can differ in the contents of the data that are stored within the client data blocks.

Recovery Consistency

A recovery-consistent state fulfills the data integrity requirements with respect to streams and ARUs, as mentioned in the beginning of Section 7.2. One requirement is that the order in which client commands within one stream are sent to LD, is respected in the recovered state. This requirement is fulfilled within a recovery-consistent state because a recovery-consistent state exactly reflects the changes of a prefix of executed committed client commands. Therefore, since LD has determined the order of execution according to the streams semantics, a recovery-consistent state respects the order in which the commands within one stream were sent to LD.

The other requirement is that, after a crash, all commands within an ARU are recovered or none of them are. LD also fulfills this requirement if it recovers to a recovery-consistent state because a recovery-consistent state reflects the changes of committed commands only. In other words, the changes made by an uncommitted ARU are not part of a recovery-consistent state. Therefore, a recovery-consistent state also satisfies the ARU requirement.

7.2.2 Recovery with a Log and Checkpoints

Now that we have discussed which state is recovered by LD after a crash, this subsection discusses the recovery technique uses by LD. For the user of any system (not just LD), the most important thing that the recovery process after a crash should do is to recover the user's precious data that were stored on disk. Actually, the system, however, has to do more than just recover a user's data blocks. The system also has to recover its own metadata since these are needed for managing the user's data blocks.

Data that must be recovered after a system failure must be stored on a medium that will survive such a failure. A hard disk is such a medium since it provides persistent (nonvolatile) storage. Furthermore, it is very reliable nowadays, which means that media failures are rare. Because of these two properties, we consider disks to be a proper medium

for storing data that must be recovered after a crash; no extra hardware is needed for recovery from system failures.

Most systems already store their valuable data on disk. However, how these data are stored on disk, that is, the method of how they are written to disk and when they are written, influences how successful a recovery process can recover the state of the disk after a crash. For example, does a system take precautions to deal with the situation when a crash occurs while data are being written to disk? The more effort a system is willing to spend on preparing for recovery from a crash, the better and/or faster its recovery after a crash can be. In practice, however, the amount of effort a system is willing to put in taking extensive precautions varies. Many systems are not willing to pay the price of the overhead involved in taking such precautions. For instance, many systems directly perform in-place updates.

One method to prepare for recovery is a combination of a log and checkpoints, which is the method that LD uses. In this method, the system uses a log in which each change to the system is logged, and additionally, the system periodically makes a checkpoint. Each relevant change to the state of the system is logged by writing a *log record* into a log. This log record describes the change that was made and contains information for undoing and/or redoing the change. A log record in LD is formed by a log tuple and any data blocks that are referred to by that log tuple, which are blocks written by the logged client command, as discussed in Chapter 5. The log records in LD contain enough information to redo the operation during recovery.

In general, the log itself is stored on some persistent medium and is often (but not necessarily) stored on a different medium than the other data of the system. Storing the log on a different medium further lowers the chance that both the log and the data themselves are lost after a media failure. Since LD only deals with system failures, it can store its log on the same disk as its client data, which is what LD does.

After a crash, the log enables the recovery process to see which changes have been made to the state of the disk. Therefore, using the information available in the log records in the log, a recovery process can decide to redo or undo individual changes in order to bring the system state into a recent and consistent state.

In order for the recovery process to recover the most recent and consistent state possible, the log on the persistent medium must be kept completely up-to-date. It is inefficient, however, to flush each individual log record to disk immediately. Therefore, LD and most log-based systems keep the head of the log, containing the most recent log records, in main memory only, and as soon as enough log records have accumulated or a certain amount of time has elapsed, they are appended to the log in one large, efficient write, and then LD starts accumulating the next log records in main memory. This accumulation of log records allows LD to efficiently keep the persistent version of the log very close to the fully up-to-date version. However, after a crash the log records that were in main memory are lost. Thus, LD's recovery process will usually not recover to the most recent, consistent state just prior to the crash, but to a somewhat earlier consistent state.

In general, logging is not used on its own in practice because recovery with only a log requires replaying the log from its very beginning. The longer the system runs, the larger the log would get and the longer recovery would take. Furthermore, if the log is the only precaution used against crashes, the ever growing log then could never be pruned

because it is needed for recovery. To solve this problem, logging is used in combination with checkpointing.

The purpose of making a checkpoint is to allow the log to be pruned. In general, there are different ways how checkpoints are used, depending on whether the log records in the log contain undo and/or redo information. In LD, the checkpoint guarantees that *all* changes made before the checkpoint are stored safely on disk. Therefore, in LD, the log records that describe the changes made before a checkpoint, are obsolete since these changes are guaranteed on disk and do not have to be redone during recovery.

Although after a checkpoint all changes described in log records in the log are already on disk, some of them are not committed yet, and must thus not be recovered after a crash. For example, during a running ARU, the log contains log records for commands of that ARU, but it does not yet contain the corresponding commitaru log tuple. If a crash occurs before the ARU is committed, then LD must recover to a state in which the effects of that uncommitted ARU are not present. The question is whether LD can prune the log if it makes a checkpoint before the ARU is committed, or does it still need the information in the log records to undo the uncommitted ARU during recovery? The answer is that to undo the changes made by uncommitted ARUs during recovery, LD does not need the corresponding log tuples. Recall that LD uses different internal logical addresses for uncommitted data (i.e., nonzero aru_ids). Consequently, LD can easily recognize which data are uncommitted and can discard them during recovery, thereby undoing the uncommitted changes. With this approach, LD's log records do not need information to undo an operation. We will further discuss LD's recovery process in Section 7.6.

Checkpointing, thus, provides LD with a tool to shrink the log once in a while. In general, however, a system using a log and checkpoints may decide to still keep an obsolete part of the log around to be able to undo any changes or keep it as an extra backup, in case that the checkpointed state would get lost due to other kinds of failures.

As mentioned before, in literature, the term 'checkpointing' is used in several different meanings and contexts. For example, in database literature, some authors use the term checkpointing to refer to the writing of a special *checkpoint-record* in the database log. This record indicates that all transactions prior to that checkpoint-record have been committed to disk and do not have to be redone during recovery. Other authors use the same term to refer to saving a consistent snapshot of a system to nonvolatile storage, such as a disk. In LD, we will use the definition in which a **checkpoint** logically represents a snapshot of an overall-consistent state.

Using both a log and checkpoints, the recovery process in LD consists of two steps. In the first step, the latest checkpointed state is recovered. This step brings LD in a consistent state, but not necessarily a very recent state. The second step consists of reading the log and redoing any changes that have been made to the state after the latest checkpoint in order to bring the system to a more recent and still consistent state. In principle, only the changes made to the state by committed ARUs must be redone. However, we choose to replay all changes in the log and afterward undo the uncommitted changes. The precise method of replaying will be discussed in Section 7.6.

Since the log has already been discussed in Chapter 5, the remainder of this chapter focuses on the checkpoint. Furthermore, we explain how the log and the checkpoint work together to enable LD to recover to the most recent, recovery-consistent state possible

after a crash by using the available data on disk.

7.3 Checkpoints in LD

In Section 7.2.2 we introduced the technique of logging and checkpointing. There we defined the checkpoint to represent a snapshot of all blocks of the disk. After a crash, the state frozen in this snapshot is used as a starting point to replay the changes recorded in the log in order to reconstruct a recent and consistent state. In LD, however, a checkpoint is a snapshot of *only* the metadata blocks. A checkpoint does not include client data blocks. Of course, client data blocks are stored somewhere on disk; they are just not part of a checkpoint. To see why it is sufficient for recovery to checkpoint only the metadata blocks, we need to look at the desired state recovered after a crash.

7.3.1 Checkpointing only Metadata

As stated in the previous section, the recovered state must be recovery consistent and recent. First, note that it is easy to transform an overall-consistent state into a recovery-consistent state. The difference between the two is merely the presence of uncommitted ARUs in the overall-consistent state. Since the changes made by uncommitted ARUs are labeled differently in LD's metadata data structures (i.e., uncommitted data are identified by logical block addresses with a nonzero aru_id), LD can easily remove the changes made by uncommitted ARUs by removing the appropriate entries in LD's data structures. This removal also includes freeing the corresponding disk blocks by recording them as 'free' in the FreeMap. These actions transform an overall-consistent state into a recovery-consistent state. Therefore, if, after a crash, LD can reconstruct an overall-consistent and recent state, LD can also recover to a recovery-consistent and recent state, and thereby, fulfill the data integrity requirements put on data blocks after a crash.

Now the issue becomes how LD can recover an overall-consistent state. Let us look at the blocks that are stored on disk immediately after a crash. Somewhere in the blocks on disk, there is already a client data state that is client-data consistent, as was argued in the previous section. Furthermore, that client data state is recent. Therefore, LD should recover to an overall-consistent state that contains this client data state. Not surprisingly, however, LD cannot immediately find this overall-consistent state on disk because the metadata state in the metadata area on this disk does not represent the same prefix of the sequence of executed client commands as the client data state does; the metadata blocks on disk are not up-to-date with the client data blocks on disk. This difference is due to LD writing metadata blocks to disk asynchronously with respect to client data blocks. Even worse, it is likely that the latest metadata blocks in this metadata area do not even contain a metadata state that is metadata consistent. Consequently, in order to reconstruct an overall-consistent state from the blocks on disk, LD must reconstruct the metadata blocks such that the reconstructed metadata state exactly reflects the changes of the same prefix of executed client commands as the latest client data state on disk reflects.

This reconstruction is exactly what LD's recovery mechanism does; it recovers a metadata state that corresponds to the client data state that was already on disk immediately before, and thus with our assumption of fail-stop semantics, also after the crash. After LD

has recovered an overall-consistent state, LD can complete the recovery process by simply turning this state into a recovery-consistent state, as stated above. Note that LD does not have to reconstruct the same metadata state that would have existed if the crash had not occurred, but only an equivalent one. A more detailed description of LD's recovery process is given in Section 7.6.

In short, LD does not have to reconstruct the client data blocks because they are already stored on disk. However, LD does need to reconstruct the corresponding metadata in order to know which client data blocks on disk make up the latest client data state that is client-data consistent, and where they are stored on disk. Therefore, recovery in LD consists of recovering LD's metadata.

7.3.2 Reconstructing the Metadata State

As mentioned before, LD uses a log and checkpoints to recover its metadata. The log contains log tuples, which describe changes to client data, and implicitly, also describe changes to LD's metadata. The checkpoint consists of a snapshot on disk of the metadata blocks at a certain moment in time. This checkpoint serves as a starting point from which LD can replay the effects that the commands in the log had on LD's metadata to advance the metadata state until it is equivalent to the state that belongs to the client data state that LD is to recover.

Note that any metadata blocks that were written after the checkpoint was made are completely ignored during recovery. This property is the consequence of the way LD recovers its metadata blocks; LD starts with the metadata blocks frozen at the checkpoint, and subsequently, applies the changes that follow from executing client commands that were logged in the log segments that have been written after the checkpoint was made. By first restoring the state of LD's metadata to the state it had when the checkpoint was made, it is as if the metadata blocks written after that checkpoint were never written. These metadata blocks are 'lost' because the FreeMap is restored to the state it had when the checkpoint was written. At that time, those metadata blocks did not exist yet, and therefore, their (future) locations in the metadata area are still marked as 'free'. Replaying the log tuples will eventually result in a metadata state that is equivalent to the one that existed somewhat prior to the crash. However, it does not necessarily mean that exactly the same metadata blocks will be written as before the crash.

For the above mentioned recovery method to function correctly, LD must ensure that the following two conditions are fulfilled:

- There must always be a client data state on disk that is client-data consistent.
- LD must be able to reconstruct a metadata state that belongs to this client data state on disk, such that the client data and metadata state form an overall-consistent state.

In Section 7.2, we already briefly argued that the first condition is fulfilled. LD fulfills the second condition with the help of a log and checkpoints. LD must ensure that *all* relevant changes to metadata since the last checkpoint are logged. Changes to metadata can originate either from executing client commands or from LD's cleaner and reorganizer processes. Chapter 5 already showed that all client commands are logged via log

tuples. Therefore, the metadata updates caused by executing these client commands are also (indirectly) recorded in the log. Changes to metadata are also initiated by LD itself by running cleaner and reorganizer processes, which move client data blocks around on disk. How LD ensures that these metadata updates do not prohibit LD from reconstructing a suitable metadata state that corresponds to the client data state after a crash, is discussed in Chapter 8, which discusses cleaner and reorganizer processes in detail.

7.3.3 Requirements of a Checkpoint

LD puts three requirements on the contents of a checkpoint and on the process of making a checkpoint:

- A checkpoint should be a snapshot of the metadata state that is *metadata consistent*, that is, the metadata state frozen in the snapshot should exactly reflect the changes of a prefix of executed client commands.
- Making a checkpoint should be *fast* and should not require much *overhead* so that normal operation of LD does not suffer much from making a checkpoint.
- A checkpoint must be persistent and must remain unchanged until the next checkpoint is made.

In LD, the state that is frozen within a checkpoint is the entire collection of LD's metadata blocks containing the most recent version of LD's data structures, at the time the checkpoint was made. The first requirement that LD puts on this state is that it must be metadata consistent. The consequence of this requirement is that data structures stored in the metadata blocks are internally and mutually consistent.

One way to make a snapshot of LD's metadata state is by writing a complete copy of all metadata blocks to (another part of the) disk. However, copying all metadata blocks may take quite some time, which is in conflict with our second requirement to keep the making of a checkpoint fast. Fortunately, it is not necessary to copy all LD's metadata blocks to make a checkpoint. Since most blocks that make up a snapshot of LD's metadata are already stored persistently on disk, LD can make a snapshot by recording only their *locations* (i.e., disk addresses).

Above we have argued that making a checkpoint involves recording the locations of all metadata blocks. Fortunately, we do not need to record the disk address of each block individually. LD's metadata consists of the following data structures: Root Mapping, Meta Mapping, Mapping, and FreeMap. Notice that the disk addresses of the blocks of the Mapping and FreeMap are already stored in the Meta Mapping. Furthermore, the disk addresses of the blocks of the Meta Mapping are stored in the Root Mapping. Consequently, the Root Mapping contains (indirect) references to (i.e., the locations of) all other metadata blocks, and therefore, it is sufficient for LD to store the Root Mapping (a single metadata block) on disk in order to make a snapshot of all metadata blocks on disk.

Besides the Root Mapping, more information must be written to disk as part of a checkpoint. Recall from Chapter 6 that the metadata information in LD's metadata data structures is spread over two data structures: the differential and the basic part. The

basic parts reside in metadata blocks on disk. Therefore, the locations of these blocks are recorded (indirectly) in the Root Mapping. However, the differential parts are in-core data structures. The information in these differential parts must also be stored in the checkpoint in order to freeze a metadata state that is metadata consistent. Therefore, they are written to disk together with the Root Mapping as part of the checkpoint. Since the differential parts are small, the total size of the data that must be written to disk to make a snapshot remains small. Consequently, writing a checkpoint adds only a little overhead, which is conform our second requirement put on checkpoints.

The last requirement states that the checkpoint must be persistent and its contents must be kept unchanged until the next checkpoint has been made. Since the checkpoint is stored on disk, the checkpoint is persistent. The contents of the checkpoint must remain unchanged for as long as the checkpoint exists because during recovery LD restores the state frozen in the checkpoint. Therefore, in general, some copy-on-write policy is necessary whenever a metadata block that is part of the checkpoint changes later on. However, since LD never overwrites blocks in place, a separate copy-on-write policy is unnecessary for LD. In the next section we look at the issue of preserving the checkpoint in more detail.

7.3.4 Preserving the Checkpoint

The metadata blocks in the snapshot that are frozen during a checkpoint must remain unchanged because they are used when LD recovers from a crash. During recovery, LD replays the commands recorded in log tuples by redoing the changes that these commands made to metadata blocks. Each log tuple describes a client command, and therefore, indirectly also describes the corresponding *changes* to LD's metadata.

In order to replay these changes to LD's metadata, LD must have access to the original metadata blocks on which the changes must be done. As a basis for these changes, LD uses the metadata blocks that were frozen during the last checkpoint, and therefore, during recovery, LD must have access to all metadata blocks that are part of that checkpoint.

Consequently, due to our choice to store only pointers to metadata blocks in a checkpoint, instead of making a copy of all metadata blocks, LD cannot simply delete metadata blocks when they become obsolete, but must preserve them, if they are part of the latest checkpoint. Note that a metadata block becomes obsolete when it is deleted or when a new version of it is written.

To illustrate when a metadata block becomes obsolete, but cannot be deleted immediately, consider the following example. Suppose that, during normal operation, a client writes a single new block of a disk file. Executing this command writes the new disk file block into LD's log together with a corresponding log tuple. Furthermore, the Mapping is updated to store the location of this new disk file block, and the FreeMap is updated accordingly. For simplicity, we will only consider the update to the Mapping, and ignore the update to the FreeMap for now. Updating the Mapping consists of reading from disk the Mapping block containing the mapping information for the corresponding disk file, inserting a new entry for the disk file block into the Mapping block, and eventually writing it back to disk at a new location. Additionally, the Meta Mapping and Root Mapping may need to be updated, but, for simplicity, we will leave them out of our discussion as

well. In principle, the old version of the Mapping block is now obsolete because a newer version of it exists. However, if the old version is part of the latest checkpoint, LD may not delete it and may not reuse its disk space yet. This old version is still necessary if LD crashes and LD has to recover by replaying its log tuples. Replaying the log tuple of the command that wrote the single new disk file block, requires access to the old version of the Mapping block in order to replay the metadata update to the Mapping recorded in the log tuple.

To ensure that the snapshot made during a checkpoint remains intact, LD preserves metadata blocks by delaying the deletion of a metadata block. To avoid overwriting a checkpointed metadata block, LD uses a kind of *shadow paging* technique [Lorie, 1977; Gray and Reuter, 1993]. Whenever a metadata block that is part of the latest checkpoint is changed, the new version of the metadata block is written to disk at another location; this technique keeps the old version intact. Since LD already uses a no in-place update policy, the contents of a (metadata) block is never directly changed in-place, and therefore, it does not take any extra effort for LD to keep a checkpointed metadata block intact other than preventing the deletion of the old version.

A remaining question is how long the contents of blocks in a snapshot must be preserved. In other words, when can LD remove the blocks of a snapshot? The answer is that LD has to wait until a next checkpoint has been made. At that time, the blocks that were part of the old snapshot, but that are no longer part of the new snapshot anymore, can be finally deleted. We will discuss how LD delays deleting metadata blocks in more detail in Section 7.9.

A minor disadvantage of this solution is that the disk space of some deleted metadata blocks cannot be reused until a following checkpoint is made. Fortunately, disk space is not a scarce resource. Furthermore, if LD makes checkpoints regularly, the number of unavailable disk blocks between two successive checkpoints is small. Even though client data blocks are not part of a snapshot, care must still be taken when deleting client data blocks in order not to affect LD's ability to recover after a crash. We will get back to this issue in Section 7.7.

7.4 Checkpoint Area

LD has reserved space on disk to hold checkpoints: the **checkpoint area**. The size and location of this area is stored in the superblock, which is at a fixed position on disk. This way, LD can always find the checkpoint area whenever LD restarts after a crash. In principle, the size and location of the checkpoint area can be changed on-the-fly when necessary (see also Chapter 8).

Similar to the log area, the checkpoint area is divided into a number of fixed-size **checkpoint area slots**. Each slot currently has a size of 128 KB and can hold one **checkpoint segment**, which represents one checkpoint (see Section 7.5). The checkpoint area must comprise at least two checkpoint area slots because LD must be able to write a new checkpoint segment while keeping the previous checkpoint unaltered. In principle, having only two checkpoint area slots is sufficient since then LD can alternate between them for writing checkpoint segments.

The disk space reserved for the checkpoint area slots is always marked as 'used' in the FreeMap. The advantage of this method is that LD never needs to update the FreeMap when checkpoint segments are written, and consequently, this method avoids metadata updates on the FreeMap during the making of a checkpoint. Overall, this method keeps the process of making a checkpoint simple (see Section 7.8).

During normal operation LD needs to remember in which checkpoint area slot it wrote the latest checkpoint in order to be able to prevent overwriting of that checkpoint segment when making a new checkpoint. LD can simply use a global variable to keep track of which slot contains the latest checkpoint. This information does not have to be directly stored on disk because, after a crash, this information can simply be recovered by scanning all checkpoint area slots and using the sequence numbers inside each checkpoint segment to find the last successfully made checkpoint.

In Figure 4.2 we showed a logical representation of the different areas on disk. For ease of reference, we have included the same figure here as Figure 7.1. Although the figure shows the checkpoint area as one contiguous area on disk, in practice, LD can spread the checkpoint area over the disk. LD can position each checkpoint area slot individually across the disk. Each slot in the checkpoint area is allocated a physically contiguous range of disk space, so that a checkpoint segment can be written into a slot with one large and, therefore, efficient disk write operation.

Figure 7.1: Logical representation of the areas on disk. This figure is a repetition of Figure 4.2.

The benefit of spreading the checkpoint area slots across the disk is that LD can make a checkpoint more efficiently. With multiple checkpoint area slots spread across the disk, LD can choose the slot that is closest to the current disk head position, when LD needs to write a checkpoint segment. This minimizes the distance that the disk head needs to travel before it can start to write checkpoint data to disk. A simple and efficient approach is to spread the number of checkpoint area slots evenly across the disk. Spreading checkpoint area slots, however, also requires more maintenance keeping track of them, especially if they are moved. Currently, LD uses only four checkpoint area slots: two on either side of the metadata area (see also Section 7.8).

7.5 Checkpoint Segment

The checkpoint itself is made by writing a **checkpoint segment** into a slot of the checkpoint area. Similar to a log segment, a checkpoint segment is written to disk with one large, efficient disk write operation. The size of a checkpoint segment is fixed and fits within one checkpoint area slot. Currently the size of a checkpoint segment is 128 KB.

The layout of a checkpoint segment is shown in Figure 7.2. The segment is surrounded by a checkpoint segment header and a checkpoint segment trailer. In between the header and trailer are the Root Mapping and the differential parts of the Meta Mapping, FreeMap, and Mapping. These various parts of the checkpoint segment are discussed below.

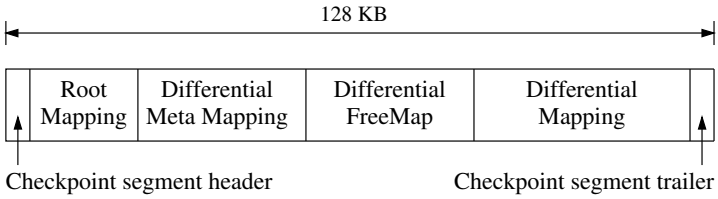


Figure 7.2: Layout of a checkpoint segment.

7.5.1 Checkpoint Segment Header and Trailer

The contents of the **checkpoint segment header** and **checkpoint segment trailer** are given in Tables 7.1 and 7.2, respectively. The header contains a **checkpoint segment identifier** or cp_segment_id, which is a 4-byte sequence number. This sequence number is used by LD to determine which checkpoint segment was the last segment successfully written before a crash.

Table 7.1: Contents of a checkpoint segment header.

Field	Description
cp_segment_id	checkpoint sequence_nr
cp_log_head	location of head of the log
cp_log_tail	location of tail of the log
cp_log_next_seg_id	log segment sequence_nr of next log segment in the log
cp_dmm_count	number of entries in the DMM
cp_dfm_count	number of entries in the DFM
cp_dm_count	number of entries in the DM

The header also contains references to the locations where the head and the tail of the log were at the time that the checkpoint was made. These locations are stored in the fields cp_log_head and cp_log_tail. After the checkpoint has been successfully made, new segments are written to the head of the log, which starts at the location referenced by the field cp_log_head. Therefore, during recovery, the cp_log_head field points to the start of the first log segment of a chain of log segments that contain log tuples, which need to be replayed in order to recover a recent and recovery-consistent state.

The cp_log_next_seg_id field contains the expected log_segment_id (i.e., log segment sequence_nr) of the first log segment of this chain of log segments. With this information, LD can recognize when it has reached the end of this chain of log segments during recovery because each log segment contains a sequence_nr. Additionally, after recovery has completed and LD has resumed normal operation, LD knows which log segment sequence_nr to use when it writes a new log segment to disk. Note that if a normal shut down was done or a crash occurred immediately after a checkpoint was successfully made, the chain of log segments that LD needs to replay during recovery is empty.

The cp_log_tail field contains a reference to the tail of the log at the time the checkpoint was made. The tail of the log refers to the oldest log segment that still contains live client data blocks, which need to be removed from the log (i.e., copied into the storage area) by a cleaner process after recovery of a crash.

The last three fields cp_dmm_count, cp_dfm_count, and cp_dm_count contain the number of entries in the Differential Meta Mapping (DMM), Differential FreeMap (DFM), and Differential Mapping (DM), respectively, which are all stored in the checkpoint segment.

The checkpoint segment trailer is used to verify the validity of a checkpoint segment. A checkpoint segment is invalid either because a media failure occurred or because it was not written completely. Since we assume that media failures do not occur, in practice, LD checks the integrity of a checkpoint segment only to verify its completeness. Using this completeness check, LD is able to write a checkpoint segment atomically. This method is similar to the way LD writes log segments (see Chapter 5).

Table 7.2: Contents of a checkpoint segment trailer.

Field	Description
cp_segment_id	sequence_nr
cp_chksum	checksum of the checkpoint segment

The checkpoint segment trailer consists of two fields: the checkpoint segment identifier and a checksum (see Table 7.2). Each field is used in a different method to check whether the checkpoint segment has been written completely. The first method is to verify that the checkpoint segment identifier in the checkpoint segment trailer matches the identifier in the checkpoint segment header. Since LD writes a checkpoint segment in one large, contiguous write, in theory, the checkpoint segment trailer is written last. Therefore, the checkpoint segment identifier in the trailer will only match the one in the header if the entire checkpoint segment has been written successfully as a whole. Checking the cp_segment_id in both the checkpoint segment header and trailer is a simple and cheap way to detect incomplete checkpoint segments.

The second method to check the integrity of the checkpoint is the use of a checksum, which is stored in the checkpoint segment trailer. The checksum currently consists of 32 bits and is calculated over the entire checkpoint segment. This second method is necessary because a disk may not always write the trailer of a checkpoint segment last, due to the use of a write cache and disk scheduling algorithms in the disk. Therefore, in order to detect incomplete checkpoint segments correctly, LD also uses a checksum.

7.5.2 Data Structures in the Checkpoint Segment

The space between the checkpoint segment header and trailer contains the following items:

- The Root Mapping
- The Differential Meta Mapping
- The Differential FreeMap
- The Differential Mapping

The Root Mapping in the checkpoint segment is a copy of the in-core Root Mapping at the time that the checkpoint is made. It contains the disk addresses of all blocks of the Meta Mapping. With the Root Mapping, LD can (indirectly) access all its metadata. Furthermore, the checkpoint segment contains the differential parts of the Meta Mapping, FreeMap and Mapping (see Chapter 6). Since these data structures are in-core only data structures and are part of the checkpointed state, they must explicitly be stored on disk during the making of a checkpoint. Therefore, storing the Root Mapping in the checkpoint segment is not sufficient; the differential parts of all LD's data structures are stored in the checkpoint segment as well.

The Differential Meta Mapping and Differential FreeMap are stored in the checkpoint segment as an array of fixed-size (operation, key, value) tuples. However, the Differential Mapping is stored differently. Recall that the value field in an entry of the Differential Mapping could either contain a physical block address (i.e., a reference to a logical block on disk) or the actual contents of a disk file header or cluster header. Since a disk file header and a cluster header are variable-size pieces of data, the Differential Mapping cannot simply be stored in a table with fixed-size entries in the checkpoint segment. In order to simplify the structure used to store the Differential Mapping in the checkpoint segment, LD stores the variable-size header information separately. The Differential Mapping is stored in the checkpoint segment in two parts: an array of fixed-size (operation, key, value) entries and an array of bytes holding all header values. Whenever an entry in the Differential Mapping refers to a header, the value field stored in the array of (operation, key, value) entries on disk does not contain the actual header data, but a reference to the place where the header data is stored in this byte array within the checkpoint segment.

The size of the header must also be stored somewhere. Storing size information of the header can be done by storing it together with the actual header data itself. Another way is to encode both the size and the location of the header in the 'value' field of the (operation, key, value) entry. Since the maximum size of a header is small, and the location can be stored as an offset within the checkpoint segment, both the location and the size of the header can be encoded together in 32 bits (it is sufficient to reserve 9 bits for the size of the header and 17 bits for its location within the checkpoint segment).

LD has reserved an amount of space in the checkpoint segment to hold the differential parts and the byte array containing header data. For simplicity, LD currently assigns static amounts of space for each differential part in a checkpoint segment. The checkpoint segment has room to store 1024 entries per differential part, and can hold a maximum of 64 KB worth of disk file or cluster headers. In order for the header data within the

Differential Mapping to fit within a checkpoint segment, LD should make a checkpoint before the amount of header data exceeds the amount of reserved space in the checkpoint segment. An alternative is to merge the Differential Mapping with the Basic Mapping, which moves the excess number of headers into the Basic Mapping, before a checkpoint is made.

7.6 The Recovery Process

Before we go into the details of how LD makes a checkpoint, we present a general discussion of the main steps that LD takes during recovery. In this section, we show how LD uses the log and the checkpoints to recover after a crash. Explaining LD's recovery process can help to understand the more technical discussion in a following section on the process of making a checkpoint.

LD's recovery process consists of the following four steps:

(1) Read the superblock.

(2) Retrieve the last successfully made checkpoint.

(3) Replay all log segments written after the last checkpoint.

(4) Abort any uncommitted ARUs.

After these steps have completed, LD has recovered to a recent, recovery-consistent state, which completes the recovery process as stated in Section 7.2.1. From now on, LD can resume normal operation and execute client commands. It is not necessary to make a new checkpoint or clean the log first. However, even though it takes a little extra time, making a checkpoint might be useful because it allows LD to perform the deletions of metadata blocks that were delayed during the replay of the log (see Section 7.9). Subsequently, LD can continue with a clean slate.

Below we discuss each step of the recovery process in more detail.

7.6.1 Read the Superblock

When LD restarts after a crash, it first reads all superblocks, and chooses the last one successfully written. Similar to the checkpoint segments, LD has multiple fixed and well-known locations where it stores superblocks on disk. LD alternates between these locations to hold the latest superblock in order to avoid in-place updates. A sequence number inside each superblock enables LD to determine which one is written last. A superblock contains information on the locations and sizes of the other areas on disk, including the checkpoint area, which is needed in the next step. Other pieces of information that LD needs from the last superblock are the logical metadata block addresses of the beginning of the FreeMap and of the root of the tree that implements the Mapping. Both addresses are also stored in the superblock.

7.6.2 Retrieve the Last Checkpoint

The second step in the recovery process is to retrieve the last checkpoint made successfully. The checkpoint area consists of a number of checkpoint area slots (currently four), which do not have to be physically adjacent on disk. The locations of these slots are stored in the superblock, which was read in the previous step. Each checkpoint area slot holds one checkpoint segment. In order to retrieve the most recent checkpoint segment, LD reads *all* checkpoint area slots. After checking the integrity of each of the checkpoint segments in those slots, LD uses the sequence number in each checkpoint segment whose integrity has been verified, to determine which checkpoint segment was written last.

This checkpoint segment contains a Root Mapping and differential parts of LD's metadata data structures, which are loaded into main memory to be used by LD. With the Root Mapping in place, LD is now able to translate logical metadata block addresses into physical block addresses. Therefore, since LD knows the logical metadata block address of the root of the Mapping, LD can now access all Mapping blocks on disk. The same holds for all the blocks of the FreeMap. In short, the data structures holding LD's metadata are now restored to the metadata-consistent state that was frozen at the time this checkpoint was made.

7.6.3 Replay Log Segments

The third step in the recovery process brings the metadata state that was restored in the previous step to a more recent, metadata-consistent state. The end result is a metadata-consistent state that belongs to the client data state that is client-data consistent, and that already existed on disk. In other words, this step brings the disk to a recent and overall-consistent state.

LD reconstructs the overall-consistent state by replaying all client commands that were executed after the checkpoint was made. The checkpoint contains a pointer to the head of the log at the time the checkpoint was made. LD starts reading log segments from this point onward. The integrity of each log segment is checked with the help of the enclosed checksum to determine whether the log segment was successfully written. If the checksum is valid, the checkpoint segment identifier in the head is checked to see if it is indeed the next in the line of sequence numbers found in prior log segments. If so, LD replays the log tuples contained in this log segment.

The log tuples are replayed in the order in which they were stored in the log segment (see also Chapter 5). Since this is the order in which the commands were executed, the recovered state incorporates the changes of these commands in the correct order.

During normal operation, the execution of an update client command in LD has two components: writing client data and updating corresponding metadata. However, replaying a command via its log tuple during recovery actually only involves performing the corresponding metadata updates. Since in LD recovery consists of restoring its metadata, only the changes to LD's metadata are relevant.

For example, consider what happens when, during recovery, LD encounters a write log tuple in the log that corresponds to a write command that wrote a single disk block of a certain disk file. This log tuple contains the logical address of the new client data block and a physical block address, that is, a reference to where the actual data block was written

on disk. Since the log tuple is on disk, the corresponding client data block is certainly also on disk. The client data block was written either within the same log segment or somewhere in the storage area if the data block was written via a direct segment.

Replaying this log tuple during recovery consists of only updating the Mapping and the FreeMap correspondingly; LD does not have to actually write the client data block itself anymore since it is already somewhere on disk. The metadata update consists of inserting the corresponding physical block address in the Mapping and updating the FreeMap by marking that location as 'used' and marking the previous location (if any) as 'free'. These updates to the Mapping and FreeMap may, in turn, also lead to updates to the Meta Mapping and Root Mapping.

Note that during the replay of the log tuple, the actual client data block might not be in the location referred to by the log tuple anymore. It may already be located in another location, or may even have been deleted. For example, such situations arise when, after sending the client command that wrote the single client data block but before the crash occurred, LD has moved the block to another location, or when another client command has deleted the block. However, in general, for each of these (future) changes to the client data block a corresponding log tuple will be in the log, which will be replayed. Therefore, in the end, LD's metadata will correctly reflect whether the client data block exists, and if so, they will hold a reference to its correct location. We will discuss the correctness of LD's recovery process in Section 7.7 in more detail.

Recall that disk file headers and cluster headers are stored in LD's Mapping. Therefore, if during this step LD encounters a log tuple that wrote a disk file header or cluster header, the log tuple is replayed by writing the header information in the Mapping, even though the header itself is also already somewhere in the log on disk.

Replaying log tuples continues until LD reaches the last log segment that was successfully written before the crash. At the end of replaying that last log segment, LD has restored a recent, overall-consistent state. Note that replaying log tuples generates only dirty metadata blocks. No new client data blocks nor any new log tuples are generated during this activity.

7.6.4 Abort Uncommitted ARUs

During the replay of log segments, LD may also encounter client commands that start new ARUs. LD replays commands that were sent within composite ARUs as normal: update commands create uncommitted versions of blocks. When LD encounters a corresponding commitaru log tuple, these uncommitted versions become the committed version by updating the Mapping accordingly.

However, when LD has replayed all log segments, it may happen that some ARUs have not been committed yet. In order to bring the disk to a recovery-consistent state without uncommitted client data blocks, LD aborts all uncommitted ARUs that still exist. This action has the same effect as executing `ld_abort_aru` on each uncommitted ARU, which removes all entries for these uncommitted ARUs in the Mapping and frees the disk space of any corresponding uncommitted blocks. The entries of uncommitted ARUs are easy to recognize in the Mapping because they have a nonzero aru_id field in their internal logical address.

At the end of this step, the recovered state is recovery-consistent, which means it contains only committed client data blocks. In other words, the state contains only the changes of client commands if the changes of all commands in the same encompassing ARU are also in that state.

Besides uncommitted ARUs, LD does not have to clean up anything else in order to undo changes that were only partially done due to a crash. For example, suppose LD has allocated disk space for a direct segment and written the direct segment to disk, but the corresponding log tuple has not yet been written to disk as part of a log segment. In this case, LD does not have to reclaim the disk space after recovery because during replay LD does not find the log tuple for the direct segment on disk, and will, therefore, not allocate the corresponding disk space. All in all, the disk space will still be correctly marked as 'free' in the FreeMap after recovery.

7.6.5 Crashes During Recovery

LD's recovery mechanism has been designed such that LD can also correctly handle a crash during recovery. During recovery, LD keeps the latest checkpoint and the log intact. Therefore, if a crash interrupts the recovery process then LD can simply restart the recovery process from the beginning again. Any data blocks that are written to disk as part of LD's recovery process are metadata blocks generated during the step in which LD replays log segments and the step in which LD aborts uncommitted ARUs. When LD writes these updated metadata blocks to disk, it does not overwrite any data block that may still be necessary for a future recovery. For example, these updated metadata blocks are not updated in-place. Therefore, when LD restores the checkpoint again, after a restart, it is as if none of these metadata blocks have been written to disk in a previous recovery attempt.

7.7 Recovery Correctness

The previous section described how LD recovers from a crash. This section argues that the recovery process, indeed, recovers to a state that is recovery-consistent. In particular, we focus on the issue whether replaying the log tuples in the log segments recovers an overall-consistent state. Although, for convenience, we will use semiformal definitions in our discussion, we will not give a complete formal proof of the correctness of LD's recovery process. The definitions are used only to enhance the readability of the arguments given in support of the correctness of LD's recovery process.

In short, LD's recovery process consists of the following four steps. The first step consists of reading the superblock to find the checkpoint area. The second checkpoint reads the last successfully made checkpoint, which can be identified by the sequence number and the checksum in each checkpoint segment. The third step, replays the log tuples found in the log segments written after the last checkpoint was made. This step reconstructs an overall-consistent state by reconstructing the metadata state. The last step, turns the overall-consistent state into a recovery-consistent state by aborting any uncommitted ARUs. Of these four recovery steps, replaying log tuples, the third step, is the most interesting step. In this section, we focus on this step.

Before we show why the third step works, we first introduce the following definitions.

- $c_0, c_1, c_2, \ldots$ represents the sequence of client commands executed by LD. We assume that these commands all refer to update client commands. Nonupdating client commands are irrelevant to our discussion since they do not change the state of the disk, and therefore, they do not affect LD's recovery.
- C_i represents the set of client data blocks resulting from the execution of the sequence of client commands $c_0, \ldots, c_i$.
- M_i represents a set of metadata blocks resulting from the execution of the sequence of client commands $c_0, \ldots, c_i$.
- l_i represents the log tuple corresponding to client command c_i. Log tuple l_i contains sufficient information to allow LD to bring metadata state M_{i-1} to M_i during recovery after a crash.

Without loss of generality, we assume that LD has successfully made the latest checkpoint after executing client command c_0. This checkpoint has frozen state M_0 on disk. Now, let us consider what happens when LD recovers after a crash. After reading the superblock, LD retrieves the latest checkpoint from disk. This step restores metadata state M_0. If we suppose that the last log tuple written in LD's log is l_n then the next step of the recovery process replays log tuples $l_1, \ldots, l_n$, which brings the metadata state to M_n. Our claim is that LD has now recovered an overall-consistent state. For this claim to be true, the corresponding state C_n must exist on disk.

To ensure that, indeed, replaying log tuples in LD recovers an overall-consistent state, LD ensures that, during normal operation, the following conditions always hold:

(1) If log tuple l_i $(i \geq 2)$ is on disk, log tuple l_{i-1} is also on disk.

(2) If log tuple l_i $(i \geq 1)$ is the *last* log tuple on disk, then client data state C_i must also be on disk.

By ensuring that these two conditions hold, LD guarantees the following:

> If, after a crash, LD recovers metadata state M_i by replaying log tuples, then the corresponding client data state C_i is also on disk.

Below we will look at each of these conditions in more detail.

The first condition states that if log tuple l_i is on disk, then log tuple l_{i-1} is also on disk. A consequence of this condition is that if log tuple l_i is on disk, so are log tuples $l_1, l_2, \ldots, l_{i-1}$. Consequently, if a crash occurs after LD has written log tuple l_i to disk, LD can recover metadata state M_i because LD can replay log tuples $l_1, l_2, \ldots, l_i$, which will advance the checkpointed metadata state M_0 to M_i.

The way LD ensures that LD can recover state M_i is by keeping the checkpoint and all log tuples written since the last checkpoint intact until a next checkpoint has been made. How LD keeps the checkpoint intact will be explained in Section 7.9. By not overwriting any log segments written after the last checkpoint, LD avoids overwriting any log tuples that are still needed to recover metadata state M_i. How LD reclaims log space is discussed in Chapter 8.

The second condition states that if log tuple l_i is the last log tuple on disk, client data state C_i must also be on disk. This condition guarantees that, if a crash occurs immediately after log tuple l_i has been written to disk, client data state C_i is also on disk. Consequently, since LD can reconstruct metadata state M_i from the checkpointed metadata state M_0 by replaying log tuples $l_1, \ldots, l_i$ (see condition 1), LD can recover to an overall-consistent state because client data state C_i is guaranteed to be on disk.

LD fulfills this condition by writing log tuples and client data to disk in a specific order that guarantees that the changes made to client data by executing client command c_i are safe on disk before or at the same time that LD writes the corresponding log tuple l_i into the log. The precise order of writing has been explained in Chapter 5. In short, the order is as follows. Before LD writes a log segment with log tuples to disk, it flushes all cached dirty client data blocks that will not be written within the same log segment, yet are referred to by a log tuple in that log segment. Such client data blocks are blocks written within direct segments or blocks that have been written by LD's reorganizer process, which will be explained in Chapter 8.

A consequence from both conditions is that, if log tuple l_{i-1} is the last log tuple on disk thus far, LD does not destroy the corresponding client data state C_{i-1}, until after client data state C_i is on disk and log tuple l_i is the last log tuple on disk. Consequently, LD can correctly recover to overall-consistent state C_{i-1} and M_{i-1} until the client data changes and log tuple for client command c_i are on disk. In other words, the transitions from one overall-consistent state to the next is atomic with respect to recovery.

However, as Chapter 5 explained, LD does not necessarily write log tuples one by one to disk, but it accumulates several log tuples in one log segment and writes them atomically to disk. Therefore, in general, if at some point in time log tuple l_j is the last log tuple on disk, subsequently writing a log segment containing log tuples $l_{j+1}, l_{j+2}, \ldots, l_i$ (corresponding to client commands $c_{j+1}, c_{j+2} \ldots c_i$) , will atomically make log tuple l_i the last log tuple on disk ($j < i$). Consequently, until that log segment has reached the disk, LD must ensure that client data state C_j is on disk since log tuple l_j is the last log tuple on disk, so that LD can recover overall-consistent state C_j and M_j.

How LD keeps the ability to recover metadata state M_j has been explained above where we discussed how LD fulfills the first condition: LD keeps the checkpoint intact and does not overwrite any of the log tuples written after that checkpoint up to and including log tuple l_j (i.e., log tuples $l_1, \ldots, l_j$).

In order to keep client data state C_j intact when LD executes the sequence of client commands $c_{j+1}, \ldots, c_i$, LD ensures that it does not overwrite nor delete any blocks from client data state C_j until C_i and log tuple l_i (together with log tuples $l_{j+1}, \ldots, l_{i-1}$) are safe on disk. In order not to overwrite blocks from C_j, LD always writes new data blocks to free spaces on disk. The free spaces on disk are indicated by the FreeMap in LD. In other words, this policy also means that LD does not perform in-place updates. Furthermore, if executing commands $c_{j+1}, \ldots, c_i$ results in deleting client data blocks from state C_j, then the actual physical deletions of these blocks are delayed, until C_i and l_i are safely on disk. Delaying deletions of client data blocks is done by delaying the corresponding updates to the FreeMap, until the log segment containing log tuples $l_{j+1}, \ldots, l_i$ and any other corresponding client data blocks have been written to disk, as has been explained in Section 5.8.2 on page 114. By delaying these updates to the FreeMap, LD prevents the

disk space of the deleted client data blocks from being reused too soon. Consequently, if a crash occurs before the log segment has been written, the contents of client data state C_j are still on disk.

7.8 The Checkpoint Process

This section discusses how LD makes a checkpoint. The main issue is how LD guarantees that the checkpoint that is made represents a snapshot of a metadata-consistent state of the disk. The answer is actually quite simple. When LD wants to make a checkpoint, it temporarily stops executing new commands and waits until all currently running commands have finished. Subsequently, LD makes a checkpoint. At that time the state is overall consistent (i.e., the metadata and client data exactly reflect the changes of the same prefix of the sequence of executed client commands), and therefore, the state is also metadata consistent.

More precisely, making a checkpoint involves the following four steps, which will be treated in more detail in subsections 7.8.1 through 7.8.4:

(1) **Prepare to checkpoint** — In this step, LD will not start executing any new or pending client commands anymore. Any newly arriving commands will be queued. However, commands that currently are already running are allowed to finish.

(2) **Flush dirty data buffers to disk** — This step involves flushing all dirty client data and metadata blocks to disk.

(3) **Assemble and write the checkpoint segment to disk** — This step writes the checkpoint segment to disk in a free checkpoint area slot. To ensure that LD can recover if a crash happens during the write of the checkpoint segment, the checkpoint segment may not be updated in-place. After this step has finished, the checkpoint has actually been made.

(4) **Delete preserved metadata blocks** — LD must preserve the previous checkpoint because it is needed during recovery. Therefore, LD delays the actual deletion of metadata blocks that have become obsolete since the making of the previous checkpoint. Now that the new checkpoint has successfully been made, LD can perform the deletion of such metadata blocks, which will allow their disk space to be reused (see also Section 7.9).

After the last step, LD can resume executing pending client commands normally. Below we look at each of these steps in more detail.

7.8.1 Prepare to Checkpoint

LD makes a checkpoint that represents a snapshot of the metadata blocks of LD. Acquiring metadata consistency is a more difficult problem when checkpoints are made on-the-fly, that is, when client commands are also being executed concurrently during the making of

a checkpoint. To avoid this complexity in our current design, we simply choose not to make a checkpoint while client commands are running.

LD cannot, however, abruptly stop already running client commands because that would leave the disk in a metadata-inconsistent state. Furthermore, LD does not support aborting individual commands that are already running. Therefore, if LD plans to make a checkpoint, LD lets already running client commands finish. At the same time, LD postpones executing any pending or newly arriving client commands. For clarity, note that LD does not have to wait until all running ARUs have finished; a checkpoint can be made in between two successive client commands of the same running ARU.

7.8.2 Flush Dirty Data Buffers

By flushing all dirty data blocks, LD ensures that the results of executed client commands are written to disk. This step includes writing all dirty client data blocks as well as all dirty metadata blocks. When all dirty blocks have been flushed to disk, the state of the blocks on disk is overall consistent, except for the information that is stored in the differential parts of LD's metadata data structures, which are kept in main memory only (see the next subsection). Flushing the dirty blocks is done in two steps:

(1) **Flush all dirty client data blocks** — In this step, all dirty client data blocks that are cached in LD's buffers are written to disk. These dirty blocks consist of the blocks of the in-core segment, the blocks of any direct segments, and the blocks that have been written as part of reorganizer or cleaner activity (see Chapter 8). For recovery purposes, the order in which these blocks are written is important. For instance, the log tuples of the blocks in a direct segment must be written only after the direct segment has been written to disk. Therefore, first all direct segments and blocks written by reorganizer or cleaner processes are written to disk in the storage area. Subsequently, the in-core segment is written to disk as a log segment including all log tuples. The result of flushing the in-core segment is that, if the checkpoint eventually succeeds, all client updates that have been committed up til now, become recoverable.

(2) **Flush all dirty metadata blocks** — In this step, LD flushes all dirty metadata blocks into the metadata area via a staccato write. Note that all dirty metadata blocks in LD's cache already have disk space allocated to them, and therefore, flushing metadata blocks to disk itself does not generate new metadata updates.

7.8.3 Assemble and Write the Checkpoint Segment

After all dirty data blocks have been written to disk, the blocks on disk are almost overall consistent. They are still not completely overall consistent because pending metadata updates may still reside in the differential parts of LD's metadata data structures, which reside in main memory only. Writing them to disk in a checkpoint segment will make the data on disk overall consistent (i.e., client data and metadata consistent), and therefore, the snapshot of the metadata state frozen during this checkpoint will be metadata consistent.

Assembling the checkpoint segment in main memory consists of the following steps. First, LD increments the sequence number, which is stored in the checkpoint segment's header and trailer. During recovery, this sequence number enables LD to see which checkpoint segment was written last. Then, LD fills the checkpoint segment header, and copies the Root Mapping and the three types of differential parts into the segment. Last, LD calculates the checksum of the checkpoint segment and fills in the checkpoint segment trailer.

In order to write the assembled checkpoint segment, LD picks an available checkpoint area slot. All checkpoint area slots are candidates, except the one that holds the previous checkpoint segment because LD may not overwrite the previous checkpoint segment in-place. Subsequently, LD writes the checkpoint segment into the chosen checkpoint area slot with one large consecutive write operation. Recall that the disk space of checkpoint area slots are not updated in the FreeMap (they are always marked 'used'); therefore, writing a checkpoint segment does not generate new metadata updates.

Currently, LD has four checkpoint slots, located two by two on either side of the metadata area. The reason to position the checkpoint slots close to the metadata area is because in the previous step LD flushes its metadata blocks to disk. Therefore, keeping the checkpoint slots close to the metadata area avoids a long expensive seek to write the checkpoint segment as the disk head is already close to the checkpoint slots.

7.8.4 Delete Preserved Metadata Blocks

When LD reaches this step, a new checkpoint has been made successfully. If a crash occurs at this point, LD will be able to recover the disk using the checkpoint segment written in the previous step. Before LD can resume executing pending client commands, however, it has to do some cleaning first.

Recall from Section 7.3.4 that LD must preserve the checkpoint, and therefore, LD delays deletions of checkpointed metadata blocks until a checkpoint is made. Since at this point in the process of making a checkpoint, the new checkpoint has been successfully made, LD can now delete any preserved checkpointed metadata blocks so that the corresponding disk space can be reused in the future. We will discuss how LD preserves checkpointed metadata blocks in Section 7.9.

The end of this step marks the end of the process of making a checkpoint. After this step, LD starts executing any pending client commands and continues as normal.

7.8.5 When to Make a Checkpoint

The algorithm for deciding when LD should make a new checkpoint can be based on a number of factors, for example:

- **Time** — a new checkpoint can be made when a certain time interval has passed since the last checkpoint. This criterion puts a limit on how old a checkpointed state can maximally be.
- **Number of log segments written** — a new checkpoint can be made when a certain number of new log segments has been written since the last checkpoint. This crite-

rion determines some maximum amount of work that needs to be replayed during recovery.

- **Number of log segments cleaned** — a new checkpoint can be made when a certain number of log segments has been cleaned since the last checkpoint. This criterion controls the amount by which the tail of the log advances. When a checkpoint is made, the tail of the log advances so that the disk space of cleaned log segments can be reused.

In principle, how frequently LD makes checkpoints influences only how fast recovery will be, not how recent the recovered state will be. Checkpointing more often will reduce the number of log tuples that need to be replayed after a crash, and therefore, decreases the average time necessary for LD to recover after a crash. The recentness of the recovered state is solely determined by how often log tuples are written to disk. However, since in our current design making a checkpoint also flushes the in-core log segment to disk, it does indirectly have some influence on the recentness of the recovered state.

Currently, our prototype implementation bases its decision when to make a new checkpoint only on the number of cleaned log segments. After a certain number of log segments have been cleaned, a new checkpoint is made which shrinks the log and frees space in the log area to be reused for new log segments (see also Chapter 8).

7.8.6 Reducing the Interruption of Making a Checkpoint

In Section 7.3 we mentioned the requirement that the process of making a checkpoint should be fast. However, all dirty buffers must be flushed to disk in order to make a checkpoint. Flushing all dirty buffers may take some time if a large number of data blocks must be written. Since during this time no new client commands are executed, this flush may have quite an impact on the response time for clients.

Fortunately, there are a few steps LD can take to reduce the noticeable interruption caused by making a checkpoint. First, client commands that only read data blocks do not update data blocks on disk, and therefore, can be executed even if LD is making a checkpoint without disturbing LD's ability to make a metadata-consistent snapshot. Only newly arriving *updating* client commands are queued. However, LD must still abide by the rules that streams put on the execution order of commands. Therefore, read commands may not overtake update commands if they are within the same stream.

Second, when the time nears that a checkpoint needs to be made, LD can enter a **pre-checkpoint phase** before it starts the four steps of making a checkpoint in which no new client commands are accepted anymore. In this pre-checkpoint phase, LD can flush dirty blocks (client data as well as metadata blocks) to disk, while still accepting and executing newly arriving update commands. The advantage of this scheme is that when LD starts the actual process of making a checkpoint, the number of dirty blocks in LD's cache is low, and therefore, not much time is necessary to write the remaining dirty blocks to disk.

7.9 Preserving Checkpointed Metadata Blocks

Recall from Section 7.3 that LD must keep the snapshot that is frozen in a checkpoint intact until the next checkpoint because LD uses this snapshot as the basis for recovery. Therefore, when, in the time that lies between the making of two successive checkpoints, client commands cause checkpointed metadata blocks to be deleted, LD delays the actual deletion of such metadata blocks. Then, as the last step of making a new checkpoint, LD performs the deletion of these metadata blocks, as has been explained in Section 7.8.4.

In LD, a metadata block is deleted in two cases. First, the metadata block could simply not be necessary anymore. For example, the sequence set of the tree that implements the Basic Mapping can shrink by merging two adjacent Mapping blocks, which deletes one metadata block. Second, a newer version of the metadata block could be written. In this case, the new version of the metadata block is physically written somewhere else on disk due to our no in-place update policy, after which the old version can be deleted. In both cases, if the deleted metadata block is part of the snapshot that was frozen during the latest checkpoint, LD may not actually delete this metadata block, but must preserve it.

7.9.1 The Metadata Block Preserve List

In this subsection we look at how LD manages to delay deletions of checkpointed metadata blocks until the next checkpoint is made so that LD can still have access to them during recovery. LD delays deletions of checkpointed metadata blocks by keeping a separate list that keeps track of all metadata blocks that should have been deleted since the latest checkpoint. This list is called the **metadata block preserve list** and contains the physical addresses of disk blocks whose contents must be preserved.

Whenever a metadata block is deleted (i.e., it has simply become unnecessary or it has been overwritten), LD does three things. First, LD marks its physical position as ‘free’ in the FreeMap. Second, LD updates the metadata block’s entry in the Meta Mapping or Root Mapping accordingly. If the obsolete metadata block is a Mapping or FreeMap block, it has an entry in the Meta Mapping; if it is a Meta Mapping block, it has an entry in the Root Mapping. The entry is deleted if the metadata block has become unnecessary; it is updated to refer to the physical position of the new version if it has been overwritten. Last, LD inserts its old physical position into the metadata block preserve list. Furthermore, whenever LD allocates disk space for a new metadata block on disk, LD always consults both the FreeMap and the metadata block preserve list before choosing and allocating a free block on disk. A physical block is chosen only if it is free according to the FreeMap *and* if it is not on the metadata block preserve list. This allocation method ensures that LD will never reuse disk space that still contains metadata blocks that are necessary during recovery. In other words, the actual physical deletion of an obsolete metadata block is delayed. Logically, however, the obsolete metadata block is gone because the FreeMap and Meta Mapping or Root Mapping have already been updated.

As soon as a new checkpoint has been made, the metadata blocks whose locations are in the metadata block preserve list can actually be deleted. Since the locations stored in the metadata block preserve list are already marked ‘free’ in the FreeMap, LD only has to clear the metadata block preserve list to make these locations reusable again. Therefore,

the last step of the process of making a checkpoint as discussed in Section 7.8 actually consists of clearing the metadata preserve list. This action finalizes the deletion of previously deleted metadata blocks, which was delayed until the next checkpoint is made.

The result of the way LD uses the metadata block preserve list is that metadata blocks are actually never freed *immediately*, but only after the next checkpoint has been made. Until that next checkpoint has been made, the deleted metadata blocks are marked 'free' in the FreeMap, but their disk space still cannot be reused, so that their contents remain intact. However, in Section 7.9.2 we will show that the disk space of some deleted metadata blocks can be reused immediately without endangering LD's ability to recover to a recovery-consistent state.

The metadata block preserve list itself does not have to be recoverable, and can, therefore, be kept in main memory. The reason is that, in case of a crash, the list is automatically reconstructed during recovery in the following way. After a crash, the last successfully made checkpoint is recovered and an empty metadata block preserve list is initialized. Next, LD replays the log, which also causes new versions of metadata blocks to be written to disk. Writing new metadata blocks, in turn, causes the old versions of these metadata blocks on disk to be deleted, and their physical addresses are inserted in the metadata block preserve list as normal. At the end of recovery, a correct metadata block preserve list will be reconstructed. In the event that a crash occurs during recovery, the recovery process can simply start again, as explained in Section 7.6.5.

7.9.2 Optimizing the Freeing of Metadata Blocks

Our current solution of not reusing disk space of deleted metadata blocks until after a following checkpoint is more strict than necessary. As explained thus far, *all* deleted metadata blocks are preserved until the next checkpoint has been made, which means that their disk space cannot be reused immediately. Fortunately, we can make an optimization.

An example where LD can delete a metadata block immediately is the following. Consider a Mapping block that is updated twice after a checkpoint. Figure 7.3 shows a graphic representation of this example. The client commands that lead to the two updates to the same Mapping block are marked in the figure as commands 1 and 2. The actual client commands are not important to the example. Furthermore, assume that prior to LD executing these client commands, a checkpoint has been made that froze a version of the Mapping block on disk. This version of the Mapping block is marked with the letter A in the figure. After command 1 is executed by LD, a new version of the Mapping block is created and eventually written to disk: version B. Version B replaces version A, but the disk space of version A cannot be reused because it is still needed for a possible recovery as it is part of the checkpoint. Therefore, the address of version A is put in the metadata block preserve list. Executing command 2 creates yet another version of the Mapping block: version C. This version replaces version B, whose disk space *can* be reused to hold new data. Even though version C may not be safe on disk yet, it is recoverable because LD can recover version C by restoring version A of the checkpoint and replaying client commands 1 and 2 using their log tuples.

The reason why in this example the disk space of version B can be reused is that LD can restore its metadata by starting with the metadata blocks frozen by the checkpoint

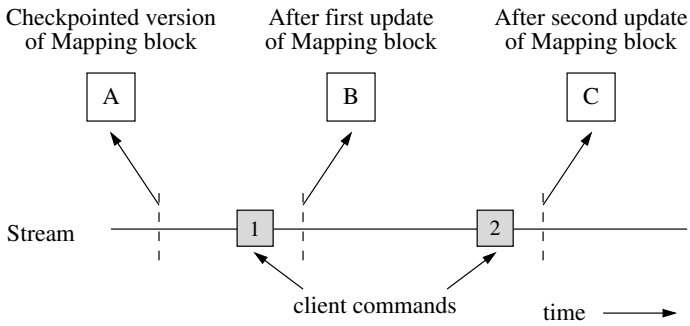


Figure 7.3: Freeing a metadata block. Mapping block A is the checkpointed version, which is used as a starting point during recovery. Client commands 1 and 2 cause the same Mapping block to be updated twice, which results in versions B and C of the Mapping block. When version B is created, version A may not be deleted. When version C is created, version B can be safely deleted and its disk space reused because during recovery version A is used to replay log tuples after a crash.

and applying all metadata changes that result from executing logged client commands. Therefore, only metadata blocks that are part of the latest checkpoint must be preserved for recovery purposes. Metadata blocks that are not part of the latest checkpoint can be safely deleted immediately and their disk space reused when necessary.

How can LD distinguish between metadata blocks that are part of the latest checkpoint and those that are not? LD could make this distinction by storing in the metadata block preserve list both the logical and the physical address of a metadata block that is to be deleted. Whenever a metadata block is deleted or overwritten, LD should always check whether the corresponding logical metadata block address is present in the metadata block preserve list. If it is not present, then this block is part of the latest checkpoint, and therefore, its logical and physical addresses are inserted in the metadata block preserve list so that its contents will be preserved until the next checkpoint, and the physical address is marked 'free' in the FreeMap. If the logical metadata block address is already present in the list, then apparently, the original checkpointed version has already been deleted or overwritten before. Therefore, this version of the metadata block that is to be deleted is not part of the latest checkpoint anymore, and consequently, this metadata block can actually be deleted, its physical address marked 'free' in the FreeMap, and its disk space reused immediately.

Nevertheless, in our current design, we use the previously described, simpler approach of not reusing the disk space of *any* metadata blocks before a next checkpoint has been made. A disadvantage of this approach is that a little more disk space remains unusable than strictly necessary, and a little more overhead results from having more entries in the metadata block preserve list. Fortunately, entries in the metadata block preserve list are small since it stores only physical addresses. Furthermore, we expect to make checkpoints often enough that the number of metadata blocks written between two successive checkpoints is relatively small so that the amount of disk space that is unnecessarily unusable is negligible and the total size of the metadata block preserve list remains small.

Chapter 8

The Storage Area

The storage area is the main place on disk where LD stores client data blocks. As expected, this area covers the largest part of the disk. LD also temporarily stores client data in its log in order to enable recovery to a consistent state after a crash as well as improve write performance. The preferred place for client data blocks, however, is the storage area, and therefore, cleaner processes move client data blocks from the log into the storage area. These cleaner processes run in the background, and preferably run when the disk is idle.

The locations of client data blocks within the storage area are important for two reasons. First, the degree of clustering of client data blocks on disk influences the overall read performance of the disk, which was explained in Chapter 2. Second, fragmentation of free space on disk influences LD's ability to find sufficiently large ranges of contiguous free space, which is important for writing direct segments, and therefore, influences the overall write performance.

In this chapter, we discuss the requirements of LD's storage area, cleaners, and reorganizers, identify problems that must be solved, and present a partial design. Since our research in this area is still on-going, the design is not complete yet. A complete design would comprise three areas: a design for the structure of the storage area, a block allocation algorithm that effectively clusters blocks, and algorithms for LD's cleaner and reorganizer processes. Instead of providing a complete design, we focus mostly on the tasks that cleaners and reorganizers must perform and present only an overview of the cleaner and reorganizer algorithms.

The rest of this chapter is structured as follows. First, in Section 8.1 we briefly look at the problem of *aging*, which deteriorates the clustering of blocks over time. Next, in Section 8.2, we present our preliminary design of the storage area, which splits up the storage area in two separate areas: the *client data large* and *client data small* areas. These two areas are the subjects of the following two sections, Sections 8.3 and 8.4. Section 8.5 deals with the problem of resizing areas, which is sometimes necessary to accommodate changes to client data over time. The last two sections in this chapter focus on LD's cleaners and reorganizers.

8.1 Problems of Aging

As mentioned before, the placement of client data blocks on disk is important for performance. Good clustering leads to good read performance, and avoiding fragmentation leads to good write performance. Good placement of client data blocks can be achieved by choosing a suitable structure for the storage area and finding effective allocation algorithms to place client data blocks within the storage area. The main problem that these structure and algorithms must overcome is aging.

Aging is the term used to describe the phenomenon that client data are not static but change continuously due to clients writing and deleting data blocks. One negative aspect of aging is the fragmentation of free space on disk, which seriously hampers the ability of the block allocation algorithm to place new blocks at locations that maintain the clustering of blocks [Smith and Seltzer, 1997]. For instance, on an aged file system, the allocation algorithm will have difficulty placing blocks of newly created files consecutively. Moreover, even the clustering of existing files on an aged file system may degrade over time. This effect occurs when existing files grow; due to fragmentation, the allocation algorithm may not be able to place the new blocks of growing files near the existing blocks of these files. It may even not be possible to place the new blocks near each other.

Fragmentation is more of a problem in systems that do not perform in-place updates, such as LD. In systems that do perform in-place updates, changed blocks are simply overwritten in-place. Consequently, the clustering of those blocks is maintained even if the file system's free space has been fragmented due to aging. The allocation algorithm in LD, however, does not update changed blocks in-place, but picks new locations for updated blocks. Therefore, fragmentation of free space makes it harder for LD to maintain the clustering of client data blocks on disk more than other systems that do perform in-place updates.

In order to guarantee sufficient clustering of client data blocks on disk all the time, the locations of these blocks on disk must be maintained dynamically, as was stated in Chapter 2. In LD, the placement of client data blocks on disk is maintained by LD's reorganizer and cleaner processes, which combat aging by restoring and/or improving clustering and lowering the amount of fragmentation in the background, which was Requirement 9, on page 42.

In summary, the design of the storage structure and the algorithms for the cleaner and reorganizer processes should realize the following goals:

- Maintain clustering — LD tries to place client data blocks in the storage area such that the clustering wishes of clients are respected, which is expected to increase read performance.

- Avoid fragmentation — LD tries to keep fragmentation of free space low. A low level of fragmentation is important for LD's write performance, and in particular, for LD's ability to write direct segments.

Furthermore, a requirement on the design of the algorithms for cleaners and reorganizers is that their overhead must be kept low. Running these processes uses up CPU cycles and requires disk I/O, which means that these resources cannot be used for the execution

of normal client commands. Therefore, reorganizer and cleaner activity should be done mostly during times in which there is no or only little client activity, and the overhead caused by running these processes must be kept to a minimum, so that clients are not hindered by them.

8.2 Structure of the Storage Area

In this section, we present a first attempt at a design of the storage area structure. The goal of such a design is twofold. First, it should enable LD to easily maintain the clustering of client blocks. Second, it should enable LD to easily write direct segments.

8.2.1 Fixed Block Locations

Traditional file systems, such as the Berkeley Fast File System (FFS) [McKusick et al., 1984], also provide an on-disk layout for blocks that tries to improve read performance by clustering blocks on disk. FFS offers clustering by allocating blocks within *cylinder groups*. A cylinder group is a contiguous range of physical disk blocks. Each cylinder group contains some file system metadata, such as i-nodes and a bitmap that keeps track of free blocks in that cylinder group, as well as user data blocks. FFS can cluster blocks by allocating them in the same cylinder group, which stores those blocks physically near each other. For example, FFS tries to allocate the data blocks of a file in the same cylinder group as its i-node. The blocks of large files, however, are spread in groups across multiple cylinder groups to avoid filling up a cylinder group with the blocks of only a single file, which denies other files in that cylinder group to grow efficiently in the future.

The disadvantage of this scheme is that, in principal, the disk addresses assigned by the block allocation algorithm to the blocks of a file are final. The file system can, therefore, not react to the aging process, which may require changes to the block layout in order to lower fragmentation. As a result, this scheme may lead to fragmentation, and therefore, a lower degree of clustering over time. Good clustering, however, is one of LD's goals; therefore, if possible, LD should monitor the block layout and improve the clustering continuously when necessary or desirable. The lack of this flexibility in the physical block layout of traditional file systems, such as FFS, is one reason why LD does not use this layout. Another reason why LD does not use the layout of a traditional file system is that LD requires some support to create free space so that it can write direct segments in order to provide good write performance. This support is not present in the storage layout as used by traditional file systems

8.2.2 Compromise 1: Segmentwise Storage

Ideally, to optimize sequential read performance, LD should provide intrafile and interfile clustering. To provide intrafile clustering, LD should store the blocks of each disk file sequentially. To provide interfile clustering, the disk files of the same cluster should also be stored sequentially. Unfortunately, maintaining such clustering is virtually impossible in a system where disk files and clusters constantly change. The more effort LD is willing to put in reorganizing the blocks on disk, the higher the degree of block clustering LD

can maintain. However, more reorganizing also implies more overhead which means less resources to execute client commands. Therefore, there is a trade-off between maintaining the desired degree of clustering and keeping the amount of overhead of maintaining that clustering low.

The first compromise we make in this trade-off is to store files segmentwise. Chapter 2 introduced the concept that files stored segmentwise on disk are considered to be sufficiently well clustered. A file is stored segmentwise on disk if all its blocks are consecutively stored in one or more segments, while the segments themselves may be spread across the disk. In Chapter 2, we also argued that good read performance can be achieved when files are stored segmentwise on disk instead of consecutively from begin to end. The degree of read performance achieved for files that are stored segmentwise depends on the characteristics of the disk and the size of the segments used. LD uses this concept of segmentwise storing for its disk files. Storing disk files segmentwise decreases the amount of effort needed to maintain the clustering on disk since these segments may be stored independently of each other while still maintaining sufficient clustering for each disk file.

8.2.3 The CDL and CDS Areas

Unfortunately, segmentwise storage of disk files to maintain good clustering is only useful for large disk files. In order to maintain good clustering for smaller files, a different storage layout must be designed. Therefore, in our design for the storage area, we distinguish two different layouts: one for large ranges of data blocks and one for small ranges of data blocks. For simplicity, in our preliminary design for the storage structure, we have given each of these layouts a separate area by splitting the storage area in two, called the **client data large (CDL)** and the **client data small (CDS)** areas. Roughly put, the CDL area holds client data that form large consecutive ranges (such as data written within direct segments, which are 256 KB each), and the CDS area holds the rest of the client data. The main advantage of splitting the storage area is that each area holds its own type of data, which makes designing a specialized storage structure and allocation algorithm for it easier. The structure and exact purpose of each of these areas is discussed in Sections 8.3 and 8.4.

Figure 8.1 shows the *physical* layout of the disk. This figure slightly differs from the *logical* layout, which was presented in Figure 4.2, on page 64. The storage area is split in two areas: CDL and CDS. In between these two areas is the metadata area. Currently, the checkpoint area of LD consists of four checkpoint slots, which are grouped two by two on either side of the metadata area (see Chapter 7). Although the figure does not clearly show it, the CDL and CDS areas are by far the largest areas on disk.

To give an indication of the relative sizes of the CDL and CDS areas, we can refer to a survey that is done on UNIX file sizes in 1993 ([Irlam, 1993]). This survey shows that close to 99% of all files in the surveyed file systems were small, that is, smaller than 256 KB, and therefore, only 1% of the files is larger. However, if we look at the amount of disk space that those files use, the numbers are quite different: the small files occupy only a third of the disk space, the other two thirds is occupied by large files. Therefore, if we assume that files smaller than 256 KB are stored in the CDS area, and larger files are mostly stored in the CDL area, this survey suggests that the CDL area is roughly twice the

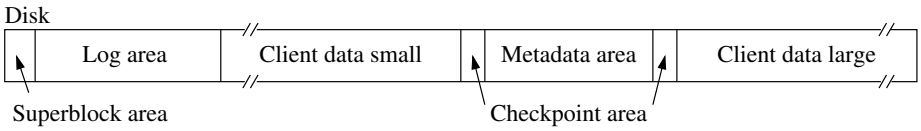


Figure 8.1: Physical representation of the areas on disk, including a preliminary design for the storage area.

size of the CDS area. The survey was taken over ten years ago. Since that time, the rise of multimedia has gone hand in hand with the need to store more large files containing photographic images, movies and sound. Consequently, we speculate that a similar survey held today may show that the division of disk space has shifted even more toward large files.

The sizes of the CDL, CDS, and metadata area may vary over time to accommodate changes to the amount of client data and metadata. LD can also react to changes in the ratio of the client data blocks forming large contiguous ranges (i.e., data which belong in the CDL area) and the rest (i.e., data which belong in the CDS area). The advantage of a variable sized metadata area is that the size of the metadata area does not impose a software limit on the amount of metadata that LD can use. Since the other areas of LD (i.e., superblock, log area, and checkpoint area) have fixed sizes, the remaining disk space is divided dynamically over the CDL, CDS and metadata area. Section 8.5 discusses the problems related to assigning disk space to each of these three areas.

For simplicity, we have chosen to keep each area physically contiguous on disk. The sizes of the CDL, CDS, and metadata areas are dynamically determined at run-time. Unfortunately, resizing the areas may involve shifting some data blocks around to keep the areas contiguous. Since the metadata area is the smallest, it is the easiest to move, and therefore, we have chosen to put the metadata together with the surrounding checkpoint area in between the CDL and CDS.

8.3 Client Data Large

The main purpose of the client data large area (CDL) is to provide an efficient storage structure to store large ranges of consecutive data, such as data written within direct segments. The structure of the CDL is as follows. The entire CDL area is divided into **CDL slots**. Each slot has a fixed size of 256 KB, which is, not coincidentally, the same size as a direct segment.

The purpose of each slot is to hold a consecutive number of data blocks that meet the following requirements:

(1) The blocks belong to the same disk file.

(2) The blocks have consecutive logical addresses.

(3) The blocks form 256 KB of data.

(4) The range of blocks starts at a logical position in the disk file that is aligned on a 256 KB boundary.

These requirements coincide with the requirements for a direct segment, which were mentioned in Section 5.7.6. Therefore, the data of a direct segment can be stored in a CDL slot. There is no restriction that only blocks of disk files for which clustering was explicitly requested by a client can be stored in a CDL slot, which leaves open the possibility that unclustered blocks can also be written as a direct segment into a CDL slot (see Section 5.7).

The size of each slot has been chosen such that good sequential read performance can be achieved when reading a logically consecutive number of data blocks that are stored in one or more slots. If more than one slot is necessary, the slots should preferably be adjoining to achieve the highest sequential read performance. However, even if the slots are randomly positioned across the CDL area, the sequential read performance is still acceptable because the overhead of the interspersed seeks is small compared to the amount of data read (see Chapter 2). Consequently, a large file can be considered to be stored sufficiently clustered if its blocks are stored in one or more slots in the CDL area.

In order to be able to write direct segments into the CDL area, the fragmentation of free space in the CDL area must be small. There are two kinds of fragmentation that LD must deal with: *external* fragmentation and *internal* fragmentation. In the CDL, external fragmentation concerns LD's ability to find multiple consecutive empty CDL slots, which depends on the distribution of empty CDL slots over the entire CDL area. Fortunately, external fragmentation in the CDL is not a problem because LD only allocates single slots. LD does not need to allocate multiple consecutive slots because clustering of data is considered sufficient even if data are stored within nonadjacent CDL slots. Therefore, there is no need to keep empty CDL slots clustered. Every time an empty CDL slot needs to be assigned, any available slot can be used, which can be found by scanning the FreeMap.

Internal fragmentation, however, may present a problem. Internal fragmentation is the amount of unused space within each CDL slot. Empty blocks within CDL slots are the result of aging. For example, when large files shrink, CDL slots with less than 256 KB of data may be created. Too much internal fragmentation results in inefficient use of the available disk space in the CDL, which, in turn, results in less available empty slots in the CDL to hold new direct segments. In time, internal fragmentation may result in LD not being able to allocate slots in the CDL to store future direct segments, which results in a lower degree of clustering.

To lower internal fragmentation, LD uses reorganizer processes to keep the CDL slots sufficiently filled with data blocks. If necessary, the reorganizer moves the contents of a CDL slot to the CDS (client data small) area when the slot is not sufficiently full anymore. The CDS area contains data that do not comply with the strict requirements for data in the CDL area. The CDS area is discussed next.

8.4 Client Data Small

The client data small area (CDS) is where the client data blocks are stored that do not meet the requirements that are put on data in the client data large area. In general, this description applies to data blocks from small (i.e., smaller than 256 KB) disk files, and to the 'tails' of larger disk files whose sizes are not a multiple of 256 KB. The problem with designing a suitable structure for the CDS is to find a structure that allows LD to efficiently store and manage small amounts of consecutive data blocks, which comprises maintaining clustering of those blocks and keeping fragmentation low.

The data in the CDS are formed by many small ranges of logically consecutive data. For example, these ranges are formed by small disk files. For read performance, the blocks of each range must be stored clustered by storing them physically close together on disk, preferable consecutively. This layout ensures intrafile clustering. Furthermore, since LD also supports interfile clustering, the disk files of the same disk cluster should preferably be stored close to each other as well. Maintaining this degree of physical clustering over time is hard because writes and deletions may continuously change disk files and disk clusters.

8.4.1 Compromise 2: The Address-Slot Table

Section 8.3 presented the first compromise between the degree of clustering and the amount of overhead to maintain such clustering; our compromise was to store large files segmentwise in the CDL area. The second compromise we make is that LD should first focus its efforts on storing individual data blocks of a logically consecutive range *near each other* in the CDS area, instead of strictly *sequentially*. Of course, the preferred clustering of blocks is to be stored sequentially, but as a suboptimal solution, we consider storing blocks in each other's vicinity also to be acceptable. With a lower priority, LD also runs reorganizer processes that, in time, will rearrange the blocks on disk to store the blocks sequentially.

To manage the blocks in the CDS, LD divides the CDS up into fixed-size **CDS slots**. The size of each slot is currently fixed at 2.5 MB. The basic idea is that LD stores blocks that must be clustered within the same CDS slot so that those blocks are physically stored near each other. In order to determine in which slot each block must be stored, LD uses the **logical-address-range-to-CDS-slot table** or **address-slot table** for short, which maps a consecutive range of logical block addresses to a CDS slot. This table indicates in which CDS slot a data block with a specific logical block address *should* reside, if it is not eligible to be stored in the CDL area. The table covers all the logical addresses in LD's Mapping, and is dynamically updated to accommodate changes in the number and sizes of disk files.

Figure 8.2 shows a simplified graphic representation of what the address-slot table does. The figure shows how the entire logical block address space, shown in the top half of the figure, could be mapped onto the CDS area with six CDS slots, depicted in the bottom half of the figure. The logical block address space is divided into consecutive ranges. A range, for example, could represent all files within a disk cluster, in which case a range could be from address (1,1,1) up to, but not including (2,1,1). This range would

represent all (committed and uncommitted) files within disk cluster 1. The blocks within a range should all be stored within a corresponding CDS slot, which is indicated by the dashed arrows in the figure. By assigning address ranges to the same CDS slot, the blocks in that range will be stored close together, and are therefore clustered. This scheme will ensure that blocks of the same disk file are stored within the same CDS slot, as long as the boundaries of the ranges are chosen at disk file boundaries.

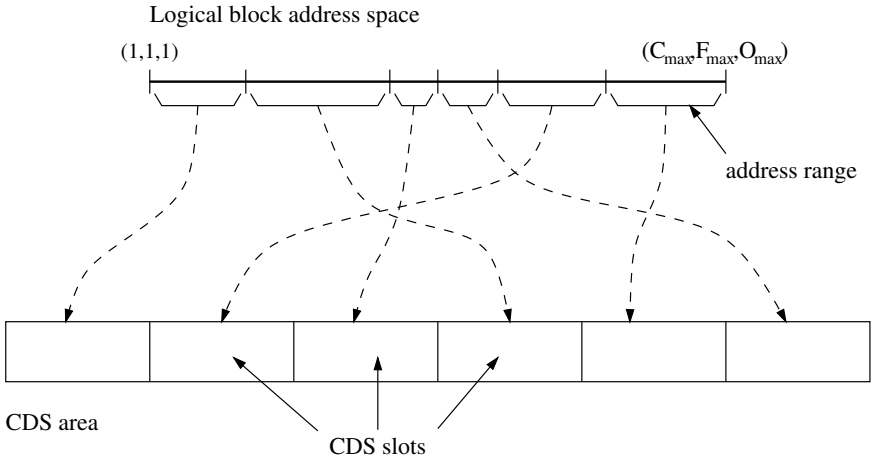


Figure 8.2: The address-slot table maps logical block address ranges onto CDS slots. The figure shows how the logical block address space is split up into consecutive ranges. A different CDS slot is assigned to each logical address range. (C_{max}, F_{max}, O_{max}) represents the largest possible logical block address.

Each entry in the address-slot table corresponds to a consecutive range of logical block addresses. These addresses refer to blocks for which a client may have requested clustering, for example, the blocks of the same disk file. If every client data block that belongs in the CDS is stored in the CDS slot indicated by the address-slot table, most client data blocks of a single disk file are stored within the same slot, and therefore, in each other's vicinity.

Two adjacent entries in the address-slot table refer to two adjacent ranges of logical block addresses. However, the corresponding slots do not have to refer to physically adjoining CDS slots, which can be seen in Figure 8.2. Therefore, in order to maintain sufficient clustering, the boundaries of the address ranges in the table entries must be chosen with care. Preferably the boundaries should lie between two consecutive disk clusters, and at least, between two consecutive disk files. This heuristic ensures that blocks of the same disk file, or even disk cluster, are mostly assigned to the same CDS slot, and are, therefore, stored physically close together. Unfortunately, it may sometimes be impossible or impractical to let the entries of the address-slot table coincide with clustering-friendly boundaries. For such cases less than optimal clustering is the result. However, since the

table has a limited number of entries and these cases can only occur between two consecutive entries, the number of occurrences in which no cluster-friendly boundaries can be found is also limited.

Due to our division of the storage area in CDL and CDS, the blocks of a disk file may be spread over the CDL and CDS. For example, this happens when a disk file is larger than 256 KB, but does not end on a 256 KB boundary. In this situation, the largest part of the disk file is stored in one or more 256 KB slots of the CDL, but the tail of the disk file, that is, the part of the disk file after the last 256 KB boundary, is stored in the CDS. Still, the disk file is considered to be sufficiently clustered since the overhead of the last seeks required to read the tail of the disk file during a sequential read is acceptable. The expectation is that LD's reorganizers will, in time, store the tail of a disk file sequentially in a CDS slot, so that only a single seek is required to read the tail of a disk file.

Another consequence of dividing the storage area into the CDL and CDS areas is that two logically sequential disk files of the same disk cluster for which clustering was requested can also be stored physically far apart. For instance, this situation occurs when one disk file is small and the other is large. The small one is stored in the CDS, whereas the large one is stored in the CDL. Reading both disk files sequentially probably leads to reasonable performance because the overhead of the seek required to start reading the large file is small compared to the amount of physically contiguous data LD can then read. However, the interfile clustering of those two disk files is not optimal, and experiments are needed to tell whether the loss in read performance is really acceptable or not.

8.4.2 Preferred Locations of Uncommitted Blocks

The address ranges in the address-slot table are specifically not expressed in terms of *internal* logical block addresses, which include the aru_ids of logical blocks. The table determines in which slots logical blocks should reside solely based on their logical block address, irrespective of whether they are written within a running ARU or not. This way, the clustering of blocks is based on their logical place within a disk file and disk cluster, even if the block is still tentative because it is written within a running ARU.

Specifying the aru_id in the address ranges of the table would lead to more client data block movements. This effect can be explained as follows. If the address-slot table used internal logical block addresses, client data blocks that are written within a running ARU would need to be moved by LD after the commit occurs. These moves are necessary because after a commit the internal logical block addresses of blocks written within an ARU change. Therefore, since the address-slot table will likely assign different CDS slots to blocks before and after a commit because their aru_ids differ, the preferred locations on disk for these blocks change. Consequently, LD's reorganizer processes will need to physically move these blocks to maintain the clustering prescribed by the address-slot table. Fortunately, if most ARUs are short-lived, client blocks written within an ARU will likely still reside in the log when the ARU commits. Therefore, it is unlikely, that LD would have to write these blocks twice, in two different CDS slots. Nevertheless, by not including the aru_id in the address ranges of the address-slot table, we avoid this problem altogether.

8.4.3 Lazy Realization of the Prescribed Layout

We stress that the table only describes in which CDS slot a data block *should* reside; it prescribes each block's *preferred* location. The table only guides the block allocation process. It does not describe where each block currently resides; that is the task of the Mapping.

LD strives to realize the distribution of blocks over CDS slots as prescribed in the address-slot table as follows. First, LD uses the address-slot table during block allocation to determine where it should write a client data block in the CDS area. Second, whenever the actual physical layout of blocks over the CDS slots is not in correspondence with the prescribed layout in the address-slot table, LD's reorganizers will correct that situation.

The reason that the actual layout of blocks does not always correspond to the prescribed layout is aging. Thus, maintaining the clustering of blocks in the CDS area is a continuous process. For example, growing disk files may have caused some CDS slots to become full. To correct this situation, LD has the ability to change the logical address ranges of the entries in the address-slot table in order to reach a more balanced distribution of data blocks across CDS slots (see Section 8.4.5). Such a redistribution of the address-slot table may also require moving (many) client data blocks to their newly assigned CDS slots, which is done by reorganizer processes (see Section 8.7).

The actual movement of blocks is done lazily in the background in order not to disturb LD's ability to serve client requests. Preferably, reorganizers run during idle periods. Furthermore, a running reorganizer is stopped as soon as possible when a client request comes in so that the client request can be served (see also Section 8.6). Until the reorganizers have restored the clustering of blocks, the actual layout of blocks differs from the one prescribed in the address-table slot.

Sometimes LD's reorganizers cannot, temporarily, keep up with restoring the correct clustering of blocks. Consequently, some CDS slots may become full, and as a result, some blocks cannot be written to their preferred CDS slot as prescribed by the address-slot table. In that case, LD writes those blocks temporarily elsewhere, which unfortunately means that the clustering of those blocks is disturbed. Later on, the reorganizers will catch up again, and move the blocks to their assigned CDS slots, which restores the desired clustering again (see Section 8.7).

The algorithm that is used to decide where to write blocks when the prescribed CDS slot is full is called the **overflow algorithm**. Currently, LD uses a simple overflow algorithm that tries to store blocks that do not fit in their prescribed CDS slot anymore, in the following CDS slot. If that slot is full as well, the one after that slot is tried, etc. To remedy the situation of full slots, LD will change the entries in the address-slot table in order to come to a more balanced distribution of client data blocks over slots, which will be discussed in Section 8.4.5. If the total CDS area is full, then the CDS area itself needs to grow, which means that the CDL and metadata areas on disk need to be resized as well, which is the subject of Section 8.5.

8.4.4 Recovering the Address-Slot Table

In order to make the address-slot table recoverable, it is easiest to store it in a checkpoint segment, which assures LD that at recovery a version of the address-slot table is available.

However, since changes to the address-slot table (see below) are not logged, these updates are lost after a crash; therefore, the recovered address-slot table may not be the most recent one. Fortunately, an up-to-date address-slot table is not crucial to the correct functioning of LD. Without the most up-to-date address-slot table, LD may decide to write data at locations that harm the clustering of blocks. Insufficient clustering, however, influences only LD's performance, but not its correctness. In time, LD will notice the imbalance in the address-slot table and take actions to create a more up-to-date address-slot table. As a result, reorganizers will restore the correct clustering of blocks again. Therefore, storing the address-slot table in the checkpoint segment is sufficient. However, since the storage area is still work-in-progress, we have not mentioned how the address-slot table is checkpointed in Chapter 7, which discussed checkpoints in LD.

8.4.5 Changing the Address-Slot Table

Each entry in the address-slot table maps a certain range of logical blocks to a particular CDS slot. The sizes of the ranges of logical blocks are determined by applying two conflicting heuristics. On the one hand, the size is chosen such that the number of blocks within that range of logical blocks fits well within the actual size of a CDS slot, leaving some room to accommodate future growth. On the other hand, the range must not be chosen too small lest the CDS slot be almost empty, wasting disk space within a CDS slot. Additionally, choosing the ranges too small unnecessarily forces blocks that are logically nearby to be stored in different CDS slots, which makes their clustering worse.

As mentioned before, even after carefully setting the logical ranges, a CDS slot may, in time, still become too full or too empty. This situation may occur because, due to aging, the actual number of client data blocks within a logical range may grow or shrink significantly. In the remainder of this chapter, we will call a CDS slot *overfull* when the number of logical blocks assigned to it is such that it leaves too little room in the CDS slot for future growth. Likewise, we will call a CDS slot *underfull* when a too small number of logical blocks is assigned to it.

When a CDS slot becomes overfull or underfull, the entries in the address-slot table must be adjusted to reach a more even distribution of logical addresses over CDS slots. LD can adjust entries in the address-slot table in two ways:

- **Split an address-slot table entry** — an entry in the address-slot table may be split into two entries, moving approximately half of the contents from the corresponding CDS slot into another empty slot.

- **Merge address-slot table entries** — N adjacent entries ($N \geq 2$) in the address-slot table may be merged into $N-1$ entries, which moves the contents of the corresponding N CDS slots into $N-1$ slots. A merge creates one empty CDS slot.

Each split and merge operation consists of two tasks. The first task consists of manipulating the corresponding entries in the address-slot table by either splitting an entry or merging two or more entries. Such a split or merge leads to a new address-slot table which represents the new desired locations of blocks within the CDS area. This first task is mostly an in-core operation and can, therefore, be done quickly.

The second task consists of moving the client data blocks on disk such that their locations correspond to the ones prescribed by the new address-slot table. These moves are done by reorganizer processes, which are responsible for making sure that client data blocks are at their prescribed locations. Reorganizers will be discussed in Section 8.7.

To avoid too many noticeable interruptions to LD's clients, the reorganizers run in the background. Consequently, as we have already mentioned, it is possible that the reorganizers are behind with their work, which means that the actual distribution of client data blocks within the CDS area is different from the one prescribed in the address-slot table. Fortunately, this situation does not influence LD's ability to execute future client requests correctly; it may, however, temporarily have a negative effect on LD's performance since the clustering of blocks is not as desired.

Next, we look at the split and merge operations in more detail.

8.4.6 Splitting Entries in the Address-Slot Table

A split is used to spread the contents of one CDS slot over two CDS slots. A split is necessary whenever, according to an entry in the address-slot table, the corresponding CDS slot is overfull, that is, when the CDS slot is required to hold too many client data blocks compared to the actual physical size of the CDS slot. Such a situation occurs, for example, when many new disk files within the logical range of an entry are created such that the total number of blocks in that logical range outgrows the actual size of a CDS slot. Note that the 'fullness' of a CDS slot is determined by the number of blocks that are assigned to that CDS slot by the address-slot table, which represents the *desired* block layout on disk. However, the actual number of blocks in each CDS slot on disk may temporarily differ from the prescribed number since the job of reaching the desired layout is done by reorganizer processes that run in the background.

The result of splitting a CDS slot is that the data blocks that must be stored in one slot are spread over two slots. For a split to be possible, an empty CDS slot must be available. If there are still empty slots available (i.e., the CDS area is only partially filled) then one of those slots can be used. Otherwise, an empty slot must be created by merging other CDS slots before a split can occur (see Section 8.4.7).

Splitting involves dividing the logical address range that is associated with the CDS slot in the address-slot table in two disjunct, contiguous ranges so that the number of logical client data blocks in each of those ranges is roughly the same. The original table entry of the overfull CDS slot is changed to refer to one of the two disjunct ranges. The other range is stored in the entry associated with the empty slot. Next, the client data blocks of the second range must be moved from the overfull slot to the empty slot, which is done lazily by reorganizer processes in the background. In Section 8.4.8, we present an example how an entry in the address-slot table is split.

LD uses a heuristic to determine when a CDS slot is overfull. LD splits an entry when the fullness of a CDS slot surpasses a certain **split threshold**. The split threshold is chosen low enough so that there is still some room left in the CDS slot beyond the split threshold because LD may not always be able to split the CDS slot as soon as its fullness has passed the threshold. However, setting the threshold too low results in too many unnecessary splits and inefficient use of the disk space, which leads to a lower degree of clustering.

The split threshold can be a static value or a dynamic value. For instance, the threshold can be fixed at 90%, indicating that CDS slots are candidates to be split when they are more than 90% full. A more dynamic method is to base the split threshold on the current average filling degree of CDS slots. The average filling degree (AFD) is calculated as follows:

$$\text{AFD} = \frac{\text{\# client data blocks in CDS area}}{\text{\# CDS slots} \times \text{size of CDS slot in blocks}} \times 100\%$$

The split threshold could then be:

$$\begin{aligned} \text{split threshold} &= \text{AFD} + \text{split-margin} \\ \text{split-margin} &= (100\% - \text{AFD}) \times \text{split-fraction} \end{aligned}$$

A CDS slot would then become a candidate for splitting when its filling degree passes a certain margin (*split-margin*) above the average filling degree. The split-margin is determined by the *split-fraction* (a value between 0 and 1), which ensures that the split-margin gets smaller as the average filling degree reaches 100%. This scheme leads to a more greedy splitting regime, which tries to keep the filling degree of all CDS slots close to the average filling degree.

Unfortunately, if only a small percentage of the CDS area is actually used, the AFD is low and CDS slots may reach the split threshold very quickly, depending on the value of the split-fraction. Therefore, to avoid splitting CDS slots that are less than half full, an additional heuristic is to split only CDS slots that are at least 50% full.

Whenever the fullness of a CDS slot is over the split threshold, the entry should be split soon to restore a more even distribution of client data blocks over CDS slots. To facilitate detecting whether the filling degree of a slot is over the split threshold, the address-slot table also keeps track of how many blocks have been logically assigned to each slot. Note that the number of blocks that actually is physically stored in each slot may be different as reorganizers may be behind with their work.

The current method to split address-slot table entries, splits one entry into two. Alternatively, LD could use a more generic algorithm that splits N adjacent entries in $N+1$ entries, which would be more in line with the method how LD merges entries. Using the more generic algorithm, LD can keep the filling degree of the new entries after a split close to the average filling degree, which will benefit a uniform distribution of the client data blocks across the CDS slots.

8.4.7 Merging Entries in the Address-Slot Table

In order to create an empty CDS slot, multiple entries in the address-slot table may be merged. Merging address-slot table entries can only be done on adjacent entries, which, therefore, refer to adjoining logical address ranges. In principle, N table entries can be merged into $N-1$ entries, freeing one CDS slot. Of course, the N entries involved in the merge must be chosen such that the number of corresponding logical client data blocks actually does fit in the space of $N-1$ CDS slots.

The merge itself consists of evenly dividing the logical address range covered by the original N entries over $N-1$ entries, while trying to put the boundaries of entries at

clustering-friendly points. Next, the client data blocks in the corresponding CDS slots must be moved to their newly assigned slots by reorganizer processes in the background. Not all blocks need moving; some blocks will remain in the same slots.

The algorithm that determines which entries to merge bases its decision on a number of heuristics. For example, merging should involve as few entries as possible to keep, the number of data blocks that need to move from one slot to another to a minimum. Another heuristic is that the slots after merging should not be so full that it is likely that at least one of them is likely to cross the split threshold, making it a candidate for a split again.

To reach a decision which entries (i.e., ranges) to merge, LD considers the fullness of each CDS slot, which is stored in the address-slot table. Similar to the split threshold, LD also has a **merge threshold**, which determines whether consecutive entries are candidates to be merged. The rule is as follows: N consecutive entries are candidates to be merged into $N-1$ entries if the average filling degree of the resulting $N-1$ CDS slots (after merging) is lower than the merge threshold. LD should then merge the range of entries that results in the lowest average filling degree after the merge and requires the least number of entries to be merged.

The merge threshold can also be a static value or a dynamic value. A dynamic merge threshold could be defined as:

$$\begin{aligned} \text{merge threshold} &= \text{AFD} + \text{merge-margin} \\ \text{merge-margin} &= (100\% - \text{AFD}) \times \text{merge-fraction} \end{aligned}$$

The *merge-fraction* must be smaller than the *split-fraction* defined in the Section 8.4.6 to avoid merges that lead to slots that are candidates for splits again. However, setting it too low means that finding entries to merge becomes harder. In the next section, we present an example in which two entries of the address-slot table are merged into one.

8.4.8 Splitting and Merging: an Example

Figure 8.3 shows an example how a table uses splits and merges to balance the client data blocks over CDS slots. The example is very simplified and assumes a CDS area with only five slots; therefore, the address-slot table only contains five entries. Furthermore, in this example, we assume that the size of each CDS slot is 1 MB, and can, thus, hold a maximum of 2048 logical blocks. For this example, we use dynamic values for the split and merge thresholds (see above) and set the split-fraction and merge-fraction at 0.5 and 0.25, respectively.

Figure 8.3(a) shows the initial situation. The first column shows the start logical address of a range. The range of an entry in the address-slot table goes up to, but not including, the address listed in the first column of the next entry. For example, the first entry covers the logical address range (1,1,1) up to, but not including, (5,1,1), which covers all disk files in clusters 1 through 4. The last entry covers all disk files in cluster 33 and higher. The second column indicates which CDS slot has been assigned to it. For convenience, the CDS-slots have been numbered 1 through 5. The last column indicates how many logical blocks are in use within the logical range for each entry.

With the numbers in the table we can determine the average filling degree, which is $\frac{6250}{5\times 2048} \times 100\% = 61\%$. The split and merge thresholds are thus 81% and 71%,

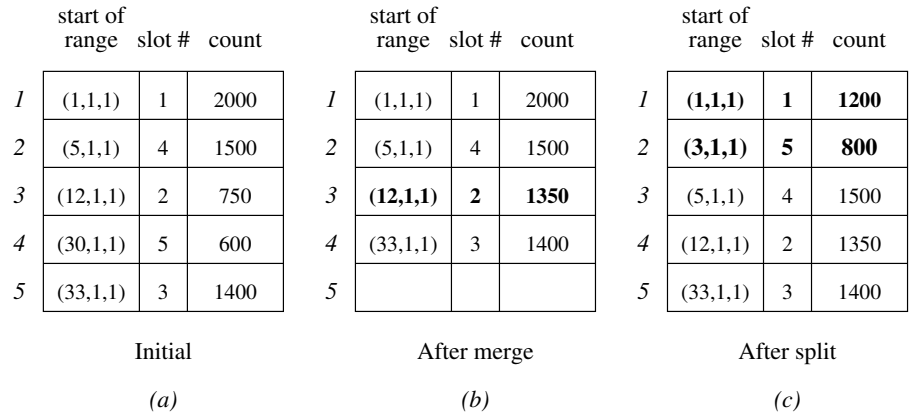


Figure 8.3: The address-slot table with five entries. (a) The first entry is full and must split. To create a free entry, the third and fourth entries are merged. (b) The address-slot table after merging the third and fourth entries, which has created an empty CDS-slot. (c) The address-slot table after splitting the first entry.

respectively. From the Figure 8.3(a) it is clear that the first entry, whose CDS slot is $\frac{2000}{2048} \times 100\% = 98\%$ full, is over the split threshold and must be split. However, since there is no empty CDS slot available, one must be created first by merging other slots.

The best entries to merge are entries three and four of the address-slot table. After merging these two entries into one, the filling degree of that slot would be $\frac{750+600}{2048} \times 100\% = 66\%$, which is below the merge threshold. Figure 8.3(b) shows the result after merging these two entries; the changed entry is printed in bold. This merge has created a free CDS slot (slot number 5). Now, the first entry can be split, which is shown in Figure 8.3(c). The entry is split in two entries; the first entry covers disk clusters 1 and 2, the second entry disk clusters 3 through 5.

8.5 Resizing Areas

In the previous sections, we introduced the CDL and CDS areas, which are responsible for storing most of LD's client data. Since the amount of client data and their division in large disk files, which are almost entirely stored in the CDL area, and small disk files, which are stored in the CDS area, varies over time, the sizes of the CDL and CDS areas must be adjusted accordingly. We, therefore, allow the CDL and CDS areas to vary in size over time. Choosing a fixed size for these areas would have put an unnecessary artificial upper limit on the number of large and/or small disk files LD can support simultaneously. Similarly, the amount of metadata also varies over time, and therefore, we also let the size of the metadata area vary over time.

To keep clustering simple, we have chosen to keep the CDL, CDS, and metadata areas contiguous on disk, in the order shown in Figure 8.1. The consequence of this decision is that resizing areas is a heavy-weight operation since enlarging one area requires shrinking

another, which potentially involves many data blocks to be moved on disk. Therefore, resizing should not be done too often.

Similar to splitting and merging CDS slots, the process of resizing areas also consists of two tasks. The first task consists of determining the new boundaries for *each* of the CDL, CDS, and metadata areas. The second task consists of implementing these boundaries by moving data blocks such that all blocks are within their designated area. This second task is done lazily by reorganizers in the background, and therefore, the process of resizing areas is not atomic, but is spread over a period of time.

Actually, these two tasks are performed independently of each other. In principle, LD could continuously determine the preferred division of disk space into the CDL, CDS, and metadata area. The reorganizers would then be continuously at work to realize the newly determined division. However, in order to lower the amount of reorganizations, in practice, LD determines a new division of disk space into the areas only in certain cases, as described below. Note that, the determined preferred division of disk space may be a moving target for the reorganizers because LD can determine a new division before the reorganizers have had enough time to realize the previous one.

In this section, we look at some problems that arise when designing algorithms to resize the areas on disk. The problems we discuss are the following:

(1) When is resizing of the areas necessary?

(2) If resizing is necessary, how does LD determine the new sizes of the areas?

(3) How does LD move the blocks on disk to conform to the new boundaries of the areas?

Below we briefly discuss each of these problems.

8.5.1 Determining When to Resize Areas

The first problem is how LD decides that a redistribution of the disk space over the areas is necessary. A simple answer is that a resize of the areas is necessary whenever an area (the CDL, CDS, or metadata area) is full and another block must be stored in it. In that case, the full area must be given more disk space, which means that the other areas must shrink a little. Since resizing areas on disk takes time, LD should preferably increase the size of an area well before it is really full. By resizing ahead of time, LD can spread the overhead of resizing over a longer period of time, which results in less noticeable interruptions to clients, which would degrade the service time for clients.

LD could use a **grow threshold** and a **shrink threshold** to determine when it is time to resize areas. If the filling degree of an area is over the grow threshold or below the shrink threshold, LD starts a resize operation and calculates new sizes for all the CDL, CDS, and metadata areas based on the current space requirements for each of these areas. The grow and shrink threshold should be set sufficiently far apart to avoid too many resize operations.

In principle, shrinking an area when it is almost empty is a good idea since this keeps the area small, and consequently, the blocks in that area are stored closer together, which

makes accessing them faster. However, the metadata area must not be set too small because LD's staccato write method needs sufficient free space to write efficiently. Furthermore, since seeks within CDL slots in the CDL area are considered to be acceptable, shrinking the CDL area to cluster the contents of CDL slots is not a high priority.

8.5.2 Determining the New Sizes of Areas

The second problem deals with determining the new sizes of the areas during a resize operation. One helpful guideline is to choose the new size of each area such that a subsequent resize operation in the near future is unlikely. This guideline will help to keep the number of resize operations low. To lower the chance of a future resize operation, each area should be given enough room to accommodate a certain level of growth of the data it holds. It is safer to give an area too much room than too little because the latter may necessitate another resize operation.

A heuristic would be to first determine the minimum amount of space needed for each area. The minimum amounts of space needed for the CDL and CDS areas are the amounts needed to actually store all data assigned to each area. The minimum amount of space for the metadata area is twice the amount of space needed to actually store all metadata. Assigning at least twice the amount allows for an efficient staccato write in the metadata area. Subsequently, each area is assigned an additional 10% of the remaining free space above the minimum amount, and the rest of the free space of the disk is divided over the three areas in proportion to their sizes. This method of space allocation ensures that each area has at least 10% of the remaining free space for growth and usually more.

Another possible guideline concerning resizing areas is to make the areas not larger than necessary. This guideline obviously conflicts with our guideline above. If the areas are chosen too large (i.e., contain too much free space), the distances between the actual data blocks in the CDL, CDS and metadata area are unnecessarily large, which results in longer seeks when accessing data. Even though long seeks are relatively cheap compared to small seeks because seek time is not linear to the length of the seek (see Section 2.1.1), it is still desirable to keep the seeks small. Therefore, a heuristic that tries to keep the size of an area small is not to make the area larger than twice the minimum size. However, this heuristic should not be used for small areas, otherwise an empty disk that is being filled would require many resize operations in the beginning.

8.5.3 Moving Data Blocks between Areas

The last problem we look at concerns the actual movement of data blocks on disk to realize the new configuration of the areas. The movement of blocks is done by reorganizer processes, which will be discussed more thoroughly in Section 8.7. Since LD considers data integrity as one of its main priorities, the way LD moves data blocks on disk must be such that a crash does not result in inconsistencies or loss of data. For this purpose, LD's reorganizers log each move operation of a client data block within the storage area.

On the other hand, recall that operations on metadata are not logged. Likewise, LD's reorganizers do not log move operations of metadata blocks. Therefore, since metadata blocks may be part of the latest checkpoint and the latest checkpoint must remain intact,

LD may not reuse the disk space of a metadata block that the reorganizer has moved from one location to another on disk. LD must wait until a new checkpoint has been made before it can reuse the disk space. Chapter 7 introduced the *metadata preserve list* to ensure that LD does not prematurely reuse the disk space of metadata blocks that are still needed for recovery. This same list is used by LD's reorganizers when they move metadata blocks as part of resizing the metadata area. As a consequence, since using the metadata preserve list implies that disk space can be reused only after a new checkpoint has been made, LD has to make a new checkpoint as part of the resizing process to complete the resize operation.

The checkpoint slots on both sides of the metadata area also have to move when the areas are resized. Moves of blocks in a checkpoint slot are not logged, but still must be done without endangering LD's ability to recover after a crash. We will discuss how the metadata area including the checkpoint slots are moved in Section 8.7.

8.6 Cleaners

Moving client data blocks on disk to maintain the clustering of blocks on disk is the task of separate processes. LD distinguishes between **cleaners** and **reorganizers**. Cleaners are responsible for moving client data blocks out of the log into the storage area. Reorganizers are responsible for all other movements of data blocks on disk, that is, all movements of blocks from the CDL, CDS and metadata areas.

Unfortunately, since cleaning and reorganizing require disk I/O, they can cause a noticeable interruption in the service of LD to clients. Therefore, cleaning and reorganizing must preferably be done during times of reduced disk traffic. Furthermore, the cleaners and reorganizers must be designed such that they perform their task in relatively small portions of work. Splitting the work in small portions means that the cleaners and reorganizers are *interruptible*, which means that LD can stop a cleaner and reorganizer quickly whenever a client requires service from LD.

The principles of spreading the work of cleaners and reorganizers in the background and making their work interruptible is important to their design. Unfortunately, since much of this work is still on-going, we cannot yet present a complete design in which these principles are implemented. We will, however, present a discussion of the different tasks of LD's cleaners and reorganizers. The remainder of this section focuses mainly on the design of LD's cleaners; LD's reorganizers are the subject of the following section.

The purpose of a cleaner is twofold. The main purpose is to clean the log so that LD can prune it when making a checkpoint. To prevent the log from growing indefinitely, the log must be pruned periodically. Furthermore, by limiting the maximum size of the log, LD can use the log area as a circular buffer. In order to prune the log, LD's cleaners must move the client data blocks that are still in use by clients from the log into the storage area. The other purpose is to restore the clustering of client data blocks that are temporarily stored in the log by moving these blocks out of the log area to their preferred locations somewhere in the storage area.

LD caches the most recently written log segments in main memory to increase the efficiency of its cleaners, The exact number of cached log segments depends on the amount

of main memory available. The advantage of caching log segments is that a cleaner can clean the log by using the blocks in main memory and writing them to disk in the storage area; in this case, the cleaner does not have to read them from disk first, which avoids extra disk accesses.

LD uses two different cleaners: a **discretionary cleaner** and a **mandatory cleaner**. Both cleaners move data blocks from the log to the storage area. The difference lies in the amount of freedom they have in choosing which blocks to move and when to move them. These cleaners are discussed in the following two subsections.

8.6.1 The Discretionary Cleaner

The **discretionary cleaner** has some freedom in choosing which blocks to move from the log into the storage area. The purpose of this cleaner is to clean the log without too much interruption to the normal service of LD; this cleaner is a lightweight cleaner. For example, the discretionary cleaner tries to move blocks from log segments that are still cached in main memory. Furthermore, it may also decide not to move certain blocks, but leave them in the log for now.

The discretionary cleaner constantly needs to deal with the trade-off between immediately moving blocks out of the log and leaving the blocks in the log as long as possible. On the one hand, by immediately moving blocks that are written in the log, LD restores their clustering, which improves the overall sequential read performance of the disk. On the other hand, leaving blocks in the log as long as possible results in less cleaning and often more efficient cleaning.

Postponing cleaning can result in less cleaning when client data blocks are repeatedly updated within a short time interval. Since the time interval in which these updates occur is short, chances are that a cleaner process has not moved the latest version of the updated block, nor any of its overwritten previous versions, out of the log yet. Therefore, since only the result of the latest update is in use by a client, the other previous versions are obsolete, and thus, do not have to be moved out of the log anymore.

Postponing cleaning can also result in more efficient cleaning because more data blocks can be accumulated in the log that must be moved to adjacent locations on disk. For example, suppose that a particular client data block has been updated and is written in the log. Subsequently, over time, one or more adjacent blocks of that particular client data block can also be updated and written in the log. In this case, the cleaner can pick all those adjacent blocks, which may have been written in different log segments, and try to cluster them in the storage area by cleaning them together. Since all blocks are selected together, the cleaner can do a better job than when it had to clean the blocks one at a time.

We can sum up a number of heuristics that the discretionary cleaner can use to decide when to leave blocks in the log instead of moving them out of the log as follows:

- If LD is temporarily very busy serving client requests, the discretionary cleaner may decide to leave blocks in the log. Moving blocks might cause a noticeable interruption of the service. Postponing cleaning activity until quieter times results in better service to clients. Of course, when client activity remains high over a long period, LD has no choice but to temporarily interrupt the service to do some cleaning when the log needs to be pruned. However, such cleaning is the task of the

mandatory cleaner (see following subsection). This heuristic can be implemented by giving the discretionary cleaner a low priority to run, which means that it will not run when LD is busy serving client requests which always run with a higher priority.

- The discretionary cleaner may give a lower cleaning priority to client data blocks for which no clustering has been requested. Restoring the clustering of such blocks is not important, and therefore, they do not have to be moved to the storage area quickly. In principle, such blocks can remain in the log longer, until free space in the log area is necessary and the log needs to be pruned.

- As mentioned before, client data blocks that are frequently updated must not be cleaned too quickly. Each new update results in a new version of that block being written in the log. The advantage of immediately cleaning that block by moving it to the storage area (i.e., better clustering) is lost if a new version is written in the log soon after it was cleaned. Therefore, leaving such blocks in the log a little longer reduces the amount of unnecessary cleaning. Unfortunately, to use this heuristic more effectively, LD needs to keep statistical information on how often blocks are updated. This information helps LD to more quickly recognize which blocks are likely candidates to be left in the log longer, and which blocks are better off when they are cleaned more quickly in order to restore their clustering, which improves the performance of future read accesses.

8.6.2 The Mandatory Cleaner

The **mandatory cleaner**, as it name suggests, has no freedom in choosing which blocks to clean. The task of this cleaner is to clean the log in order to shrink the log. The mandatory cleaner always cleans the log starting at its tail (the most recently written log segments are at the log's head). It takes one or more log segments from the tail of the log and cleans them by moving all live client data blocks in those segments into the storage area. The live client data blocks can be identified by consulting the FreeMap. Since the tail of the log is usually far behind its head, it is unlikely that some of those log segments are still cached in main memory. Therefore, the mandatory cleaner will usually have to read its segments from disk first.

In principle, when LD can keep up with serving client requests and has enough time for cleaning, the mandatory cleaner has little work to do since the discretionary cleaner will have cleaned most log segments already. However, if the discretionary cleaner cannot clean the log quickly enough, the mandatory cleaner must do its job, and it will interrupt client requests when necessary.

8.6.3 Making Cleaning Recoverable

Both reorganizing and cleaning involve moving client data blocks from one place to another. Note, however, that neither activity changes the contents of client data; they only move data blocks on disk. Although LD expects that clients can indirectly notice the effects of data block movement in improved read and write performance, the movements

themselves are invisible to clients because LD supports location transparency (see Chapter 2).

Since clients cannot tell when blocks are moved on disk, LD does not necessarily have to guarantee that the work of its reorganizers and cleaners is recovered after a crash. Indeed, clients are mainly interested in the contents of their data, and not their exact locations on disk. Of course, LD must still guarantee that after recovery the state of the disk contains the changes of a prefix of executed committed client commands (i.e., recovery consistency), but only with respect to client commands. The results of reorganizer or cleaner activity (i.e., block movement) do not have to be in that state.

If it is not too difficult, however, we do want to recover the work that LD's reorganizers and cleaners have done, so that, after a crash, the work does not have to be redone. LD uses two different methods to make the work of its reorganizers and cleaners recoverable. Each method influences the moment when LD can delete the client data block at the old location after it has been moved to another location by a reorganizer or a cleaner process.

In this subsection, we discuss how LD ensures that the work of cleaners is made recoverable; reorganizers are discussed in Section 8.7. The work done by cleaners must be done such that it survives crashes and it may not interfere with LD's ability to recover to a recovery consistent state, as was explained in Chapter 7.

After a cleaner has moved a client data block out of the log, the block at the old location in the log may not be deleted (i.e., its disk space may not be reused) until the move of the client data block into the storage area has become replayable. Until the move is replayable, LD, after a crash, will recover to a state which still contains a reference to the block in the log.

One way to ensure the move is replayable is to let LD's cleaner processes record their actions in the log with log tuples. This approach is similar to the logging of other client commands, which are made recoverable by log tuples. Note, however, that there is something paradoxical about this approach: in order to create free space in the log, new data (i.e., log tuples) must be written in the log.

We have, therefore, chosen a simpler approach in which a cleaner does not log its actions in the log. Instead, LD waits until a checkpoint has been made before it reuses the log space freed by a cleaner's actions. After a checkpoint has been made, all changes made to client data blocks by cleaner processes, including all corresponding changes to LD's metadata, are safe on disk. Therefore, at that time, LD can reuse the log space that contained client data blocks that have been moved by cleaner processes.

If LD regularly makes checkpoints and the cleaning activity can keep up with the speed at which the log grows, the amount of free space in the log area will not be a critical resource. In this approach, the work done by a cleaner is only recoverable after a checkpoint has been made. The cleaning work done after the latest checkpoint is, however, lost in case of a crash. Fortunately, as explained before, the loss of cleaner work is not important for recovery from a correctness perspective.

8.6.4 Shrinking the Log

In this section, we illustrate how cleaning shrinks LD's log. LD keeps track of the head and the tail of the log. LD writes new log segments at the head of the log, and a cleaner

process shrinks the log starting at the log's tail. Since LD uses the log area as a circular buffer, LD must prevent that the head of the log overtakes the tail of the log, otherwise LD will overwrite a part of the log that is still in use.

The way LD keeps track of which client data blocks have been copied out of the log by a cleaner process, but may not be reused until the next checkpoint is as follows. When LD makes a checkpoint, LD stores a pointer to the tail of the log in the checkpoint segment header (cp_log_tail). This position indicates the point up to which new log segments can be written. Past that position, a cleaner may have already moved client data blocks out of the log into the storage area, but those actions are not recoverable yet, and therefore, that part of the log may not be reused. When a new checkpoint has been made, the tail of the log can be moved forward up to the end of the last log segment that a cleaner has cleaned completely, making this part of the log reusable. This method is simple and efficient because LD does not have to keep track of each individual block that a cleaner has moved; only a single pointer to the tail of the log is kept.

Figure 8.4(a) shows a graphical representation how the checkpoint and the log are related. The top of the figure shows the log area on disk. The log area is partly filled with log segments, as shown in the figure by the gray area. Beneath the log area we have shown a simplified checkpoint segment, representing the latest, successfully written checkpoint segment. It contains two pointers: one to the head of the log and one to the tail of the log. However, those pointers point to the head and tail of the log at the time that the checkpoint was made. Since then new log segments may have been written, which has moved the head of the log forward, as shown in the figure.

The mandatory cleaner process may also have progressed since the previous checkpoint was made. Let us suppose that LD's mandatory cleaner has cleaned a number of log segments starting at the tail of the log and its current location is as shown in the figure. Note that the cleaned segments, represented in the figure by area 1, may not be reused because the work done by LD's cleaners is not yet recoverable. The cleaned log space becomes reusable only after a new checkpoint has been made. Also note that if a crash occurs in this situation, the log segments within area 2, that is, the log segments that were written after the latest checkpoint, must be replayed during recovery. Furthermore, LD's cleaner processes must clean the log segments in area 1 again since LD has not recovered the work of its cleaners.

Figure 8.4(b) shows what happens to the head and tail of the log after a new checkpoint has been made. As part of making a new checkpoint, the in-core log segment has been flushed to disk, which has moved the current head of the log forward. Furthermore, the tail of the log has also moved forward to the current position of the mandatory cleaner. These new positions of the head and tail are written in the new checkpoint segment as shown in the figure.

8.6.5 Copying Data Blocks into the Storage Area

A remaining point of discussion is to what locations LD's cleaners move the client data blocks from the log. What algorithms do the cleaners use to determine the new locations of those blocks in order to restore their clustering with other blocks of the same disk file, if clustering was requested at all? In principle, blocks for which no clustering has been

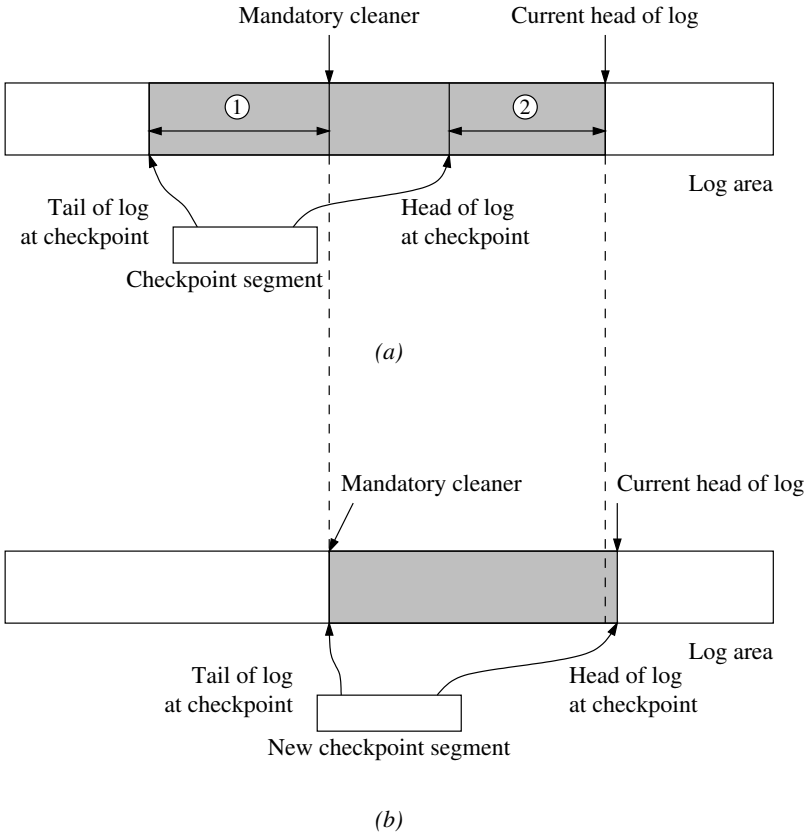


Figure 8.4: The relationship between a checkpoint and the log. (a) The situation in which a checkpoint has been made in the past. The part of the log area that is filled with log segments is shown as a gray area. The checkpoint segment contains pointers to the head and tail of the log at the time that the checkpoint was made. The current head of the log on disk has already moved forward. At the same time, the mandatory cleaner has also moved forward by cleaning some log segments. The log space in area 1 may not be reused until after the next checkpoint. Area 2 contains the log segments that must be replayed after a crash. (b) The situation after a new checkpoint has been made. The tail of the log has moved forward to the current position of the mandatory cleaner. Making a checkpoint involves flushing the in-core segment, which advances the head of the log slightly.

requested could be dealt with differently. However, currently, LD does not make this distinction, and uses the same algorithms to deal with all blocks. Below we discuss the heuristics that these algorithms use.

LD starts by checking to see if a sufficient number of other blocks in the log can be found to form a segment that complies with the requirements to fill a CDL slot. If so, the cleaner assembles these blocks together, allocates a free CDL slot and writes those blocks into the slot.

If LD is unsuccessful in finding a suitable location, LD determines whether the block that is to be moved out of the log has a previous location where it was stored already clustered with other blocks. Recall from Chapter 6 that LD uses a differential technique, which stores recent updates to LD's metadata structures separately in differential parts. Since the block in the log has been recently written, its mapping entry is likely still in the Differential Mapping. Therefore, by checking the Basic Mapping, LD can discover whether that block had a previous location. If so, LD can check whether that previous position is still available, and whether it is physically close to its logically adjacent blocks. If so, the cleaner can move the block in the log to its previous location, overwriting its old value, which restores the clustering of that block.

This heuristic does not always correctly reveal whether the block had a previous location because it depends on whether the most recent update is still in the Differential Mapping and has not been merged into the Basic Mapping. However, in cases where a few blocks of an otherwise stable disk file have changed, this method will likely keep the disk file stored clustered on disk, without too much effort.

If these steps do not find a suitable location, the block must be stored somewhere in the CDS area. Consequently, LD consults the address-slot table to find the CDS slot in which the block belongs. It then allocates a free block in the corresponding CDS slot and writes the block in that slot. The allocation algorithm can try to find an address that is physically close to the locations of that block's neighbors to increase the clustering as much as possible. If there is no room in the CDS slot anymore, the overflow algorithm is used, which will store the block in another CDS slot.

8.7 Reorganizers

The main purpose of LD's reorganizer processes is to maintain the clustering of the blocks in the storage area. In addition, reorganizers are also used to resize the CDL, CDS and metadata areas. We distinguish between different reorganizer processes because we can distinguish different reorganizer tasks that need to be done. In practice, a single reorganizer process could perform all these tasks, but for simplicity we refer to each task as a separate process.

The job of the reorganizer processes is to move blocks that are 'out of place'. Blocks are considered out of place in a number of situations. Below we list these situations in decreasing order of severity. First, blocks are out of place if they are located in the wrong area due to a recent resize operation, which has changed the boundaries of the areas such that some blocks are now in the wrong area. Second, Section 8.3 listed requirements that determine whether client data blocks belong in the CDL or CDS area. Client data blocks

are considered to be out of place if they currently reside in the CDL area, but according to the requirements in Section 8.3 should be located in the CDS area, or vice versa. Third, in the CDS area, client data blocks are out of place if they are not within their assigned CDS slot as prescribed by the address-slot table. Fourth, client data blocks within a CDS slot are considered out of place if the blocks are not stored perfectly clustered. Section 8.4 mentioned that LD should first focus on storing client data blocks within their assigned CDS slot. With a lower priority, however, LD can try to store the blocks within a CDS slot perfectly clustered, which means that all blocks of each disk file within the address range assigned to a CDS slot are stored consecutively, and all disk files are stored in order.

Note that the reorganizers themselves do not use their own criteria to determine when blocks are out of place; that is already determined by the address-slot table and by splitting the disk up into the CDL, CDS, and metadata areas. The reorganizers only have to make sure that the blocks are at their prescribed locations.

Given our preliminary design of the storage area, which divides the area into the CDL, CDS, and metadata areas, we can sum up the different tasks necessary to maintain the correct clustering as follows.

- **CDS shrink** — If, during a resize operation, the newly determined size of the CDS area is smaller than before, the old CDS area must shrink by moving the blocks in CDS slots that lie outside the new CDS area to locations that do lie within the new CDS area.
- **CDL shrink** — If, during a resize operation, the newly determined size of the CDL area is smaller than before, the old CDL area must shrink by moving the blocks in CDL slots that lie outside the new CDL area to locations that do lie within the new CDL area.
- **Metadata resize** — If, during a resize operation, metadata blocks lie outside the newly determined metadata area, these blocks must be moved to locations that do lie within the new metadata area.
- **CDL move-out** — If client data blocks in a CDL slot do not comply with the requirements imposed on data in the CDL area anymore (see Section 8.3), these blocks are moved from the CDL to the CDS area. This task will create empty slots in the CDL area, which can hold new direct segments.
- **CDS move-out** — If client data blocks in a CDS slot meet the requirements imposed on data in the CDL area, these blocks are moved from the CDS to the CDL area. This task creates free space in the CDS and clusters those blocks in the CDL area where they can be maintained easier.
- **CDS slot move-in** — The address-slot table determines for each CDS slot which blocks should be located in this slot. This task concerns making sure that each client data block lies in its assigned CDS slot. Blocks that lie in another CDS slot are moved to their assigned CDS slot.
- **CDS slot reorder** — The clustering of blocks within a CDS slot can be optimized by making sure that all blocks of the same disk file are stored sequentially within

the CDS slot, and that the disk files themselves are stored sequentially, as well. This task is a low priority task.

All these tasks move blocks on disk. To make these moves recoverable, most of these reorganizers log their actions via log tuples. Only the metadata resize task does not write log tuples when it moves metadata blocks. When it moves metadata blocks, it uses the metadata preserve list, as explained in Chapter 7 and checkpoints to make its work recoverable. Below we will describe each task in some more detail.

8.7.1 CDS Shrink and CDL Shrink

The first three reorganizer tasks have similar purposes. They are responsible for moving data blocks so that, after a resize operation, the blocks of each area are within that area's new boundaries. Each of the three tasks is responsible for moving data blocks of one of the corresponding area within its new boundaries. This subsection discusses the CDS shrink and CDL shrink tasks. The metadata resize task is discussed in the next subsection.

Data blocks of the CDL or CDS area need only be moved if the corresponding area has shrunk so that a few CDL and/or CDS slots have come to lie outside their area. If the CDL or CDS area is assigned a larger area, the new area always encompasses the old area, and therefore, no data blocks of that corresponding area need to be moved. No data need to be moved due to the way LD physically arranges the CDL, CDS, and metadata area on disk, which also explains the absence of the reorganizer tasks 'CDS grow' and 'CDL grow' in addition to their shrink counterparts.

The CDS shrink task consists of scanning the Mapping and finding the blocks that have come to lie outside the new boundaries of the CDS area. Moving the blocks inside the area consists of consulting the address-slot table to determine into which CDS slot the blocks need to be moved. If the slot is full, the overflow algorithm mentioned in Section 8.4 is used; eventually, other reorganizer tasks will correct the situation of overfull CDS slots (see below).

The process of finding blocks that lie outside the new boundaries of the CDS area can be considerably improved if LD keeps a *reverse index*, which keeps track of which data blocks are in each slot. Unfortunately, maintaining a reverse index for CDS slots, such as storing 'summary information' in each slot, involves quite some overhead. Therefore, since resize operations are expected to be rare and the actual movement of client data blocks is done in the background during idle periods, we have chosen not to include a reverse index for now. Experiments have to show whether scanning the Mapping is too costly.

The CDL shrink task moves entire CDL slots. It uses the Mapping to identify the CDL slots that lie outside the new boundaries and subsequently moves their contents to other empty CDL slots that lie within the new boundaries of the area. Here, too, a reverse index for the blocks in the CDL slots can improve performance. Since each CDL slot contains a consecutive range of blocks of a single disk file, the reverse index could simply be a table of (CDL slot number, logical start address) entries, in which the logical start address would be the logical block address of the first block of the range of blocks stored in that CDL slot. The advantage of such a reverse index on segments only (not individual blocks) is that it requires less effort to maintain and is smaller in size.

8.7.2 Metadata Resize

The metadata resize task moves individual metadata blocks of the metadata area whenever the metadata area boundaries have moved. The Meta Mapping is scanned to find the metadata blocks that lie outside the metadata area, and are subsequently moved into the metadata area and their old locations are entered in the metadata block preserve list. This task includes moving checkpoint slots. Resizing the metadata area and moving the checkpoint slots is done in two steps. Each step consists of moving one side of the metadata area, including the two checkpoint slots at that side.

An interesting question with moving a side of the metadata area is how LD can move the corresponding checkpoint slots without interfering with LD's ability to recover to a recovery consistent state. Moving the checkpoint slots on one side of the metadata area involves the following actions:

(1) The new locations to which the checkpoints slots are going to be moved are cleared, which involves moving live blocks (client data blocks or metadata blocks) that are located at these new locations to other locations.

(2) The old locations of the checkpoint slots that are going to be moved are marked as 'free' in the FreeMap.

(3) A new checkpoint is made. This checkpoint is necessary to make the disk space of the cleared locations reusable if they contained metadata blocks. The checkpoint slot that is chosen to hold the checkpoint segment that is written during the making of this checkpoint is located at the *other* side of the metadata area, that is, the side of the metadata area that is currently not being moved.

(4) The checkpoint slots at the new locations are zeroed to ensure that they do not hold a valid checkpoint segment by accident. Furthermore, their locations are marked as 'used' in the FreeMap.

(5) A new superblock, which records the new locations of the checkpoint slots, is written to disk.

After the last action, LD has moved the checkpoint slots on one side of the metadata area, and it can use them to hold future checkpoint segments. To verify that this algorithm is correct, let us consider what happens when a crash occurs during the move of one side of the metadata area. If the crash occurs somewhere before step 3, LD has not made a checkpoint yet. Therefore, when LD recovers, it restores the state stored in the latest checkpoint, which still exists. The changes to metadata blocks in steps 1 and 2 are not logged, and therefore, are lost. This includes any updates made in step 2 to LD's FreeMap, and moves of any metadata blocks in step 1. If client data blocks were moved in step 1, then the corresponding log tuples have been written to disk, which enables LD to replay the moves. Since LD has not overwritten the old locations of any of the blocks it has moved in step 1 yet, LD can recovery successfully.

Now, let us consider what happens if a crash occurs before step 5, but after step 3. In this case, when LD recovers, it restores the state frozen in the checkpoint made in step 3. In this state, the locations of the old checkpoint slots are already marked 'free'

in the FreeMap. However, the superblock still denotes the old locations as being the current checkpoint locations. LD corrects this situation by marking the current checkpoint locations as 'used' in the FreeMap again. The general rule is that the superblock indicates which locations are used as checkpoint slots, and these locations must always be marked 'used' in the FreeMap. Therefore, during recovery, LD always marks the checkpoint locations as denoted in the superblock as 'used' in the FreeMap.

The new checkpoint locations may or may not have been zeroed in step 4, depending on whether the crash occured before or after step 4, but their locations are correctly marked as 'free' in the FreeMap after recovery because no checkpoint has been made after step 4. All in all, LD can recover correctly from a crash that occurs before step 5.

Last, let us suppose that a crash occurs after step 5, but before a new checkpoint has been made. In that case, LD has already updated the superblock. Since the new locations were zeroed in step 4, during recovery, LD will recognize that these new checkpoint slots do not contain valid checkpoint segments and will recover the checkpoint frozen in step 3. The new checkpoint slots are still marked 'free' in the FreeMap, but this inconsistency will be corrected during the recovery process, as was mentioned before; during recovery, the checkpoint slots mentioned in the superblock are marked 'used' in the FreeMap, if they are not already marked so.

Note that these actions focus on moving the checkpoint slots. Of course, any blocks that lie in the area between the old and new position of the side of the metadata area must also be moved out of the way. These blocks are either client data blocks or metadata blocks. In the former case, these blocks are moved by the CDS shrink or CDL shrink task. In the latter case, these blocks are moved by the metadata resize task.

8.7.3 CDL Move-out and CDS Move-out

The following two reorganizer tasks are meant to move client data blocks between the CDL and CDS areas. The CDL move-out task looks for CDL slots whose blocks do not meet the criteria for blocks in the CDL area anymore, and therefore, these blocks should be moved into the CDS area. The CDL move-out task moves each block in such a CDL slot into that block's corresponding CDS slot which can be found by consulting the address-slot table. This task guarantees that the CDL area has enough empty CDL slots to hold new direct segments.

The CDS move-out task deals with moving blocks in the other direction, that is, from the CDS area to the CDL area. Sometimes, disk files slowly grow large enough so that they become eligible to be stored in the CDL area. If scanning the Mapping reveals such a disk file, then LD forms a direct segment from that disk file's blocks and moves it into an empty CDL slot.

8.7.4 CDS Slot Move-in and CDS Slot Reorder

The last two reorganizer tasks deal with moving data within the CDS area itself. Recall that LD can change the address-slot table whenever there is an imbalance in the distribution of data over CDS slots. For instance, this is necessary when, due to normal client

activity, some address ranges grow more than proportionally. This imbalance can be counteracted by splitting and merging CDS slots.

After a split or a merge, the client data blocks must be moved to their new CDS slots, which is what the CDS slot move-in task does. This task visits each entry in the address-slot table in turn in a round robin fashion. For each entry it checks whether any blocks in the address range are located in the wrong CDS slot. If so, these blocks are moved into the assigned CDS slot, if possible. If this CDS slot is full, the CDS slot move-in task stops processing this CDS slot and continues examining the next entry in the address-slot table. After the task has processed the last entry, it starts processing the first entry again. By running this algorithm continuously, eventually, all blocks will reside in their correct CDS slot, at least if the disk is idle often and long enough.

The CDS slot move-in task only accomplishes that all blocks reside somewhere within their assigned CDS slots. However, to really optimize the clustering of the blocks within a CDS slot, the blocks of the same disk file need to be positioned sequentially. This reorganization is done by the CDS slot reorder task. Its job is to examine each CDS slot in turn, and try to reorder the blocks in that slot such that the blocks of the same disk file are laid out sequentially. However, the CDS slot move-in task is a low priority task. Storing the blocks within their corresponding CDS slot so that they are stored near each other has a higher priority.

8.7.5 Importance of Reorganizations

In this section, we briefly look at the importance of the reorganizers. What is the effect on LD if the reorganizers, temporarily, cannot keep up with the work generated by client commands? Unfortunately, it is difficult to give an exact answer since data from real experiments are not available yet. However, we can make some observations that can help estimate the implications of reorganizers for LD.

The first observation we can make is that whether the reorganizers can keep up with their work or not has no influence on the correct functioning of LD. LD can always serve client requests correctly. The performance of LD, however, is influenced by the degree of clustering of data blocks on disk and the fragmentation of free space. Therefore, the more the reorganizers are behind with their work, the more the performance may degrade.

To get some idea of the effect of reorganizers on the performance of LD, we have to look at the likelihood that reorganizers will not be able to keep up with their work, and at the possible consequences if they cannot. Whether the reorganizers can keep up with their work depends on the amount of work they are required to do. In the following subsection, we look at the different types of work that must be done. We can roughly distinguish among three types: reorganization after a resize operation, reorganization after a split or merge in the address-slot table, and reorganizations that move blocks between the CDL and CDS area.

Reorganizations after a Resize Operation

The CDS shrink, CDL shrink, and metadata resize tasks are required as part of a resize operation. Fortunately, the first observation we can make concerning these reorganizer

tasks is that resize operations are expected not to occur very frequently. Especially when the disk is not close to being full, the areas on disk can be chosen such that they have ample of room to grow. Therefore, if clients do not change their behavior patterns too much, the distribution of data blocks over the CDL, CDS and metadata areas will not fluctuate very much, which means that resize operations are only sporadically required.

However, if a resize operation is required, what is the amount of work that the CDS shrink, CDL shrink, and metadata resize tasks must do? This amount depends on how much the new situation differs from the old situation. The larger the resize of an area, the more likely it is that more data blocks must be moved on disk. Fortunately, another observation we can make is that shrinking an area can often be done with large disk reads and writes; blocks often do not have to be moved individually.

For example, shrinking the CDL area consists of reading entire CDL slots that fall outside boundaries of the new (smaller) CDL area and writing them to new CDL slots as a whole. Unfortunately, when the CDS area shrinks, moving whole slots is not always possible. If a CDS slot falls outside the new boundaries of the CDS area, the data blocks of that slot cannot always be written in one single new CDS slot together, as the address-slot table may have changed. This change could be such that the contents of the CDS slot that falls outside the new CDS area must be spread over two (or maybe even more) CDS slots after the resize. Furthermore, the CDS slot may also contain blocks that do not even belong in that CDS slot, but have been temporarily written there by the overflow algorithm. Such misplaced blocks must be moved to their assigned CDS slot individually or in small groups when possible. Metadata blocks in the metadata area can also be more or less written in large units. The metadata blocks outside the metadata area can be read in order of their position on disk and written in the new metadata area via the staccato write method.

Reorganizations after a Split or Merge Operation

The other type of work for reorganizers is generated after a split or merge of entries in the address-slot table. After slots in the address-slot table have been split or merged, it is the task of the CDS slot move-in reorganizer to move the client data blocks of the affected CDS slots to their newly assigned CDS slot. We can make the following two observations. First, each split or merge operation only effects the blocks within the corresponding CDS slots; therefore, the number of blocks that potentially needs to be moved is relatively small. Second, even though according to the new address-slot table, some blocks are now out of place because they are in the wrong CDS slot, their clustering is actually *not* disturbed; they are still stored close together, just in the wrong CDS slot. Therefore, the read performance of LD does not really suffer that much if the reorganizers do not move the blocks immediately.

For example, consider a disk file that is stored in CDS slot 1, and suppose that after splitting the corresponding entry in the address-slot table, the disk file is now assigned CDS slot 8. Consequently, the disk file is now in the wrong CDS slot, and must be moved by a reorganizer to the correct CDS slot. Note, however, that the disk file is still clustered.

Now consider what happens when new blocks of the same file are written before reorganizers had time to move the disk file to CDS slot 8. These new blocks must be

clustered with the other blocks of the same disk file. Unfortunately, the new blocks are placed according to the new address-slot table, and therefore, end up in CDS slot 8, far away from the rest of the disk file, which is still in CDS slot 1. This situation will remain so until the reorganizers have completed the split operation by moving the corresponding blocks from CDS slot 1 to slot 8. Fortunately, if the reorganizers can keep up with their work, the time necessary to complete the split operation is not very long.

Reorganizations between the CDL and CDS Areas

The last type of reorganizations we look at are formed by the CDL move-out and CDS move-out tasks. The CDL move-out task moves the contents of a CDL slot that do not fulfill the requirements imposed on data in the CDL area to the CDS area. The CDS move-out task moves data blocks from the CDS area to the CDL area whenever they satisfy the requirements to be stored in a CDL slot.

The requirements have been chosen such that, in general, large disk files (i.e., larger than 256 KB) are mostly stored in the CDL area; only the tail of a large disk file may be stored in the CDS area. Small files are stored in the CDS area. Therefore, in general, the CDL move-out or CDS move-out tasks have work to do when the tail of a file shrinks or a file grows beyond a 256 KB boundary.

Analysis of file system access patterns, however, have shown that most files are written as a whole [Ousterhout et al., 1985]. Therefore, if we assume that such files are stored on disk via disk files, then most changes to files are passed to LD by rewriting the whole corresponding disk file. Subsequently, depending on the size of the disk file, LD writes them either into the CDL area via direct segments or in the log. In the latter case, cleaners will eventually write them into the CDS area. In short, for such files, the CDL move-out and CDS move-out reorganizers do not have to come into action. Only if disk files grow or shrink by writing or deleting individual or small ranges of blocks, is it possible that growing or shrinking disk files must be moved from the CDS area to the CDL area, or vice versa.

What happens when the CDL move-out and CDS move-out reorganizers cannot keep up with their work? In that case, the clustering of blocks is not disturbed since the blocks are still stored within the same CDL slot or CDS slot; they are just stored in the wrong area. Therefore, the read performance will not suffer much. However, large disk files in the CDS area may lead to overfull CDS slots, and small disk files in the CDL area raises the degree of internal fragmentation in the CDL area. Therefore, in extreme cases, the write performance of LD may suffer because LD may have to use the overflow algorithm to write blocks in the CDS area, and LD may not be able to find empty CDL slots to write direct segments. Fortunately, if the disk is not filled to capacity and the sizes of the areas have been chosen such that there is enough free space, LD can afford to let the CDL move-out and CDS move-out reorganizers be behind with their work. However, this supposition assumes that their is enough idle time in the near future in which the reorganizers can catch up.

8.8 Some Open Issues

In the beginning of this chapter we already stated that our research in this area is still on-going. The preliminary design we discussed above already covers all major parts of a complete solution. However, many points are still unclear and the effectiveness of the design can only be proved with experiments. In this section, we look at some important open issues that need to be addressed in further research.

One open issue concerns the actual algorithms and thresholds that we use for splitting and merging the entries in the address-slot table, as well as for resizing the areas on disk. We have only given some heuristics and guidelines that can be used in designing the algorithms. However, a detailed design is missing. Unfortunately, the success of actual algorithms can be proven best by validating them in experiments. Preferably, the experiments should test how the algorithms perform under real-life workloads and not synthetic workloads.

Another open issue deals with the amount of overhead of the cleaners and reorganizers. LD relies on the cleaners and reorganizers to maintain the clustering of the blocks. If disk traffic is rather bursty, then LD can run its cleaners and reorganizers in the idle periods (e.g., see [Baker et al., 1991; Ousterhout et al., 1985]). In that case, the cleaners and reorganizers will likely be able to keep up with normal activity.

However, if traffic is less bursty than expected or required, LD's cleaners and reorganizers may need to run while LD is serving client commands. It is, therefore, important that the overhead of cleaners and reorganizers remains small, and their work is interruptible, otherwise LD's performance may suffer too much in periods of heavy disk traffic. In Section 9.6, we will come back to the issue of how much idle time is required for LD's reorganizers to run in the background.

The last open issue we mention here is that LD currently does not make a distinction between disk files for which clustering has been requested and disk files for which clustering has not been requested. In our preliminary design, LD tries to cluster all blocks. It is likely that by not clustering blocks for which clustering is unimportant, some efficiency can be gained. However, algorithms need to be designed to determine where to store such unclustered blocks.

Chapter 9

Experiments

Chapters 3 – 8 discussed the design of LD in great detail. In the final chapters of this dissertation, we evaluate the design of LD. The evaluation of LD focuses on LD's design goals. Recall that the goals of LD, which have been presented in Chapter 3, are threefold: to improve the modularity of storage management software, to improve the support for data integrity, and to provide performance competitive to other systems.

This evaluation is done in three chapters. The first of these, the current chapter, evaluates the design of LD experimentally. The second chapter, Chapter 10: *Related Work*, discusses other storage systems, and compares their features to the features of LD. The last chapter, Chapter 11: *Summary and Conclusions*, ends this dissertation with a summary, some conclusions, and a look at future directions for LD.

In this chapter, we evaluate how well LD has reached its design goals experimentally. To evaluate LD, we built prototypes of both LD and a file system (LDFS) that uses it, and ran performance measurements on them. These measurements test the system's ability to create, write, read, and delete files. The results from these measurements are compared to the results obtained from running analogous measurements on other systems.

The structure of this chapter is as follows. Section 9.1 introduces the type of experiments that we performed. Section 9.2 discusses the setup of our experiments. It describes the test method we used to run our experiments. Furthermore, it presents the prototype implementations of LD and LDFS, the file system built on top of it. Finally, it describes the other file systems that were evaluated in our experiments. Section 9.3 describes the results of the measurements we conducted that are related to modularity. Section 9.4 describes the measurements we ran to evaluate LD's performance under various circumstances. The results of these measurements are presented in Section 9.5. Section 9.6 presents arguments that support our assumption that reorganizer processes can do their work in the background. Section 9.7 discusses some performance problems with LD's metadata performance, and presents possible solutions. Section 9.8 closes this chapter with a summary.

9.1 Introduction

Our aim is to evaluate the current design of LD by examining how well it improves modularity, improves data integrity, and how well it performs compared to other file systems. Ideally, we would run separate experiments to examine LD's behavior in each of these areas in a real-life setting. Unfortunately, it is difficult to determine how well LD has reached its goals, and particularly, its first two goals: improved modularity and improved data integrity.

The effectiveness of improved modularity can be seen in practice only when LD is used to implement different storage systems. Currently, however, only a single client system exists that uses LD; this client is LDFS, a file system on top of LD. Therefore, it is too early to draw conclusions about how well LD improves the modularity of storage systems. However, in Section 9.3 we will briefly look at the prototype implementations of LD and LDFS and make some preliminary observations in this area.

LD's improved data integrity, LD's second goal, is also difficult to evaluate in an experimental setting because improved data integrity is a quality aspect, which is not expressible in numbers. LD's data integrity features can best be evaluated by comparing them to the data integrity features provided by other storage systems under various system failure scenarios. Therefore, a more detailed evaluation of LD's improved data integrity is postponed until Chapter 10 in which a number of storage systems are compared to LD on a number of design issues. In this chapter, we discuss only an experiment that measures how fast a file system using LD can recover after a crash in comparison to other file systems. This experiment will be discussed in Section 9.4.7, and the results will be presented in Section 9.5.7.

LD's goal with respect to performance, on the other hand, can be evaluated experimentally. The aim of LD's design is to provide performance competitive to other storage systems. In order to evaluate whether LD has achieved this goal, we have done a performance comparison between LD and other storage systems. This comparison has been done by running a number of performance measurements, or experiments. Preferably, experiments should be run that measure LD's performance under real-life workloads. Unfortunately, our current prototype is not suitable to run real-life performance measurements (see Section 9.2).

Moreover, it is difficult, and maybe not even feasible, to determine which set of experiments reflects a real-life workload sufficiently. Many parameters determine the characteristics of a workload: the ratio of read versus write commands, the pattern in which these commands are given, the amount of time between commands, the amount of data that is read or written, etc. Defining a workload then consists of choosing a set of suitable values for these parameters. Unfortunately, in real life there are so many different workloads that any single set of values is inadequate to sufficiently test LD's real-life performance. To adequately evaluate LD's real-life performance using real-life workloads, we need to find a large number of sets of values for the parameters, each representing different real-life workloads. Determining which workloads to use would be very arbitrary and would depend on the relative emphasis placed on different workload types, including business, research, multimedia, web-surfing, and other types. In addition, building and validating all these workloads would be a great deal of work.

Therefore, we have chosen to run a number of measurements that are based on synthetic workloads and focus on specific performance aspects of LD and other systems. More precisely, these measurements individually test a system's create, delete, read, and write performance under varying conditions. The advantage of using such measurements is that each one isolates one aspect of LD's performance, which allows us to identify and analyze the strong and weak spots of LD more precisely. The disadvantage is that such measurements do not reliably predict LD's behavior in more realistic situations because each one focuses only on one aspect. However, these measurements do provide us with some insight in how well LD performs under varying circumstances compared to other existing systems. Additionally, since we know the performance of the other systems in practice, these measurements give us some indication how well LD would perform in a real-life setting.

9.2 Setup of the Experiments

Before we describe the experiments in more detail, we first describe the environment in which we ran the experiments and the method we used to perform them. With the experiments, we want to measure the performance of LD and compare it to the performance of other storage systems. Since LD is a low-level data storage system, we should compare LD's performance to the performance of other low-level data storage systems. Unfortunately, such low-level systems are not as readily available for testing as other more higher-level storage systems, such as file systems. Therefore, we chose to implement a prototype of both LD and a file system layer on top of LD, called the **LD File System**, or simply **LDFS**, and compare the performance of the (LD + LDFS) combination to the performance of several other file systems, such as FFS, Ext3, and ReiserFS (see Section 9.2.5 for a discussion of all tested file systems).

The prototypes of both LD and LDFS have been implemented as libraries that run within user space (see also Sections 9.2.2 and 9.2.3). In the future, LD must be integrated into a kernel as it deals with low-level storage aspects. However, for simplicity, the first prototypes of LD and LDFS have not yet been integrated into a kernel, but are separate libraries. A second implementation of an in-kernel prototype has also begun. Unfortunately, despite our best efforts, we were not able to implement this in-kernel prototype in time (see also Section 9.2.2), and therefore, we were forced to use the library versions of LD and LDFS in our experiments.

The advantage of the prototypes implemented in user space is that they can be developed and debugged more easily than kernel versions of LD and LDFS. A disadvantage, however, is that performance results obtained on the user-level versions of LD and LDFS cannot be directly compared to performance results obtained on other systems that do run within a kernel, such as file systems. We, therefore, devised a test method that allows us to compare the results of running performance measurements on the (LD + LDFS) combination and other regular file systems, even though the former runs in user space and the others run in kernel space. This test method will be described in Section 9.2.1 below.

In our performance measurements, we looked at four performance aspects of each file system:

(1) How fast can it create new (empty) files?

(2) How fast can it write data into files?

(3) How fast can it read data sequentially from files?

(4) How fast can it delete existing files?

We evaluated each file system on these four performance aspects by performing test runs on each tested file system that created, wrote, read, and deleted a number of files on an aged file system. These test runs were repeated on each file system several times, while varying the number of files that were created and the amount of bytes that were written into these files during a test run. The performance measurements are discussed in more detail in Section 9.4.

9.2.1 Test Method

The experiments consisted of running performance measurements on the (LD + LDFS) combination and on several other file systems. Unfortunately, making a fair comparison was not straightforward as we used different operating systems (FreeBSD and Linux) in our experiments. More importantly, since the prototype implementations of LD and LDFS ran in user space, it was unfair to compare their performance directly to the performance of other file systems that were all integrated into the kernel. Process scheduling effects, context switches, and memory-to-memory copies between user space and kernel space would certainly prevent LD and LDFS from reaching a performance comparable to the performance of file systems integrated in the kernel. Therefore, comparing the real-time performance of LD and LDFS directly to other file systems would not be informative, as it would be comparing apples and oranges.

The difference between testing a file system in user space and a file system in kernel space is illustrated in Figure 9.1. The test driver in the figure runs experiments to test the underlying storage system. We will explain the experiments in more detail in Section 9.4. Figure 9.1(a) shows how the prototype implementations of LD and LDFS were tested. Both LD and LDFS ran in user space. LD, in turn, used the I/O interface of the underlying operating system, indicated by the disk driver in the figure, which sent its commands to the disk. Figure 9.1(b) shows how file systems that ran in kernel space were tested. An example of such a file system is FFS. The test driver still ran in user space; the tested file system, however, ran in kernel space.

To make a fair comparison between the performance of the (LD + LDFS) combination and the performance of other file systems, we used a test method that focuses only on disk traffic, ignoring the time spent in context switches and memory-to-memory copies. Focusing on disk traffic alone allowed us to examine how many disk accesses a file system generated and how effective it utilized the disk's bandwidth under a particular measurement. Leaving CPU activity out of our comparisons will still yield credible comparisons under the assumption that disk traffic is the bottleneck in storage systems. We expect disk traffic to remain a bottleneck in the near future since technology advancements in the field of CPUs exceed the advancements in the field of disks (see also Chapter 2). Therefore, instead of measuring the real-time performance of the file systems when running our

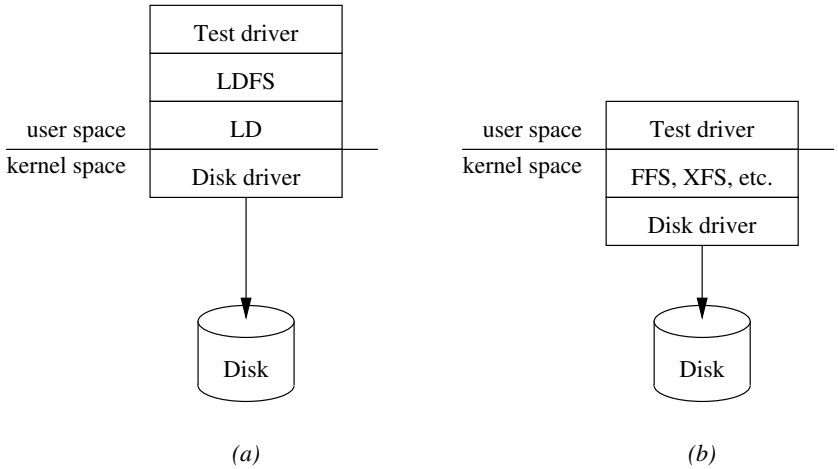


Figure 9.1: Running experiments. (a) The prototypes LD and LDFS run in user space. (b) Other file systems run directly in kernel space.

experiments, we chose to concentrate on the disk accesses that each system performed during each experiment.

Our test method was as follows. Each experiment consisted of two phases. In the first phase, the test driver ran a performance measurement by sending commands to the tested file system. This phase is shown in Figure 9.2. The details of which commands were send to the file system in each experiment are presented in later sections. In short, these experiments created, wrote, read, and deleted files by calling the appropriate interface functions of the underlying file system. We ran each experiment on all of the tested file systems, such as FFS, XFS, or the combination of (LD + LDFS).

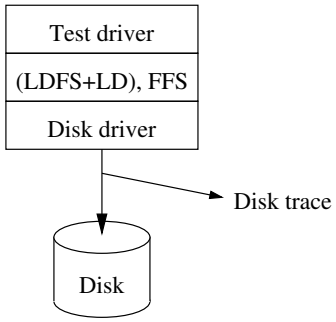


Figure 9.2: The disk trace generation phase of an experiment: a disk trace is generated while the experiment runs.

The tested file system, in turn, called routines of the disk driver which sent out the corresponding requests to the underlying disk. For our experiments, we had modified the disk drivers in the FreeBSD, NetBSD, and Linux kernels in such a way that they logged each command that they sent to the disk. For the Linux kernel, we used an existing disk tracing tool called 'Linux Disk Trace Buffer' [Linux Trace] to log each command. These logged commands formed a **disk trace**. Each entry in the disk trace consisted of a tuple (operation, address, length). The operation was either a *Read* or a *Write* operation, indicating a disk operation that read data blocks from disk or wrote data blocks to disk, respectively. The address contained the starting physical block address of a range of blocks on disk from which data was read from or written to. The last field contained the number of sectors that the range of blocks comprised. In the remainder of this chapter, we will refer to this phase as the **disk trace generation phase**.

In the second phase of an experiment, the disk traces obtained in the first phase were subsequently used as input to a trace driver which sent the commands in the disk trace directly to the disk via the disk driver. The disk then performed the read and write commands as fast as it could. This second phase is depicted in Figure 9.3. The measurement results from timing this second phase are used in our discussions to compare the performance of the tested file systems. In the remainder of this chapter, we will refer to this phase as the **disk trace execution phase**.

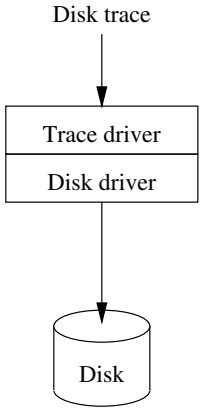


Figure 9.3: The disk trace execution phase of an experiment: the disk trace is fed to the trace-driver.

Although, in principle, we could have used the above mechanism to generate the disk trace for the combination of (LD + LDFS) during the disk trace generation phase of an experiment, we generated LD's disk traces differently. Instead of generating LD's disk trace from within the modified operating system, we let LD generate its own traces. Recall that LD is a library that runs on top of a normal file system. Whenever LD issued requests to read or write blocks from the underlying file system, it also wrote a trace entry in the form of an (operation, address, length)-tuple. The advantage of letting LD generate its own traces is that LD could include meta-information within the trace stating the reason

why data were being read or written. For example, LD could output that it was in the process of making a checkpoint or flushing a log segment. This information was helpful as debugging information and helped us to explain the performance results. Other than the extra debugging information, LD's disk traces were similar to the disk traces generated for other file systems.

9.2.2 LD Library

The prototype of LD implements the interface functions as presented in Chapter 3. The prototype, written in the C programming language, is implemented as a library and runs in user space. Client applications that use LD, such as LDFS, are simply linked to the LD library. The library uses the I/O interface provided by the underlying operating system to access the disk.

Differences between Design and Prototype

The prototype implements most of the features of LD as described in previous chapters. The implementation of the prototype, however, does deviate from the original design in some areas, mostly for ease of implementation. The five areas in which LD's prototype differs from its design are briefly discussed below.

(1) **User-level implementation** — We already mentioned that LD has been implemented as a library that runs in user space. It has not yet been integrated into a kernel. This difference means that the experiments on LD will be influenced by context switches, memory-to-memory copies, and process scheduling issues. Consequently, the performance of LD will not be directly comparable to other systems that have been integrated into a kernel.

Of course, we would have much preferred to use an in-kernel implementation of LD in our performance measurements. Unfortunately, a first attempt to integrate LD into a FreeBSD kernel did not result in a stable kernel, which was probably due to a bug in the kernel. When faced with the choice to use the user-level implementation or to start another integration attempt using a newer version of the FreeBSD kernel, time considerations forced us to use the user-level implementation.

(2) **Single stream only** — The prototype implementation of LD does not support concurrent access via multiple streams. Currently, only a single stream is supported. Therefore, in our experiments, we did not test LD's performance under concurrent accesses. Our performance measurements only have a single thread of control.

(3) **No read-ahead** — The design of LD also included support for efficient read-ahead, as described in Chapter 3. However, this functionality is currently missing in our prototype. Therefore, we did not use or test the read-ahead features of LD.

(4) **Eager discretionary cleaner** — The prototype implementation of LD has both a discretionary cleaner and a mandatory cleaner (see Section 8.6). By design, the discretionary cleaner has freedom in choosing which blocks to move from the log into the storage area, and which blocks to leave in the log for the mandatory cleaner to

clean up. However, in our prototype, the discretionary cleaner uses an eager cleaning algorithm, which means that the discretionary cleaner cleans all log blocks before they are purged from the cache. We will discuss this topic further in a separate paragraph below.

(5) **No background cleaners and reorganizers** — The design of LD uses cleaners and reorganizers that are implemented as separate processes which are able to run independently of each other. Using separate processes allows the cleaners and reorganizers to run in the background, preferably when disk traffic is low. However, the prototype implementation does not use separate processes for cleaners and reorganizers. Instead, the cleaner and reorganizer tasks are programmed within the LD library and are programmed to run when predetermined conditions are fulfilled. For example, the eager discretionary cleaner runs just before cached log segments are purged from the cache, as we will see below. Additionally, the reorganizers of LD run only when they are explicitly started by hand. In our experiments, the reorganizers are run outside the experiment, that is, *between* two performance measurements, and therefore, the overhead of reorganizing is not measured. The only exception is when, during an experiment, either the CDS, CDL, or metadata area runs out of free space. In such a case, the areas are resized immediately, which means that the disk activity involved in the resize is logged in the disk trace, and therefore, the overhead of resizing the areas is also in the performance result (see also Section 9.4.5).

Eager Discretionary Cleaner

Chapter 8 introduced two cleaners: the *discretionary* cleaner and the *mandatory* cleaner. The discretionary cleaner has freedom in choosing when to run and which blocks to move from the log into the storage area, whereas the mandatory cleaner does not. The mandatory cleaner cleans starting at the oldest log segments, and therefore, moves the tail of the log forward, creating new empty log segments.

In a real-life setting with a final implementation of LD, we expect that, during normal activity, disk traffic is sufficiently bursty that a discretionary cleaner has enough opportunities to clean most of the live data from recently written log segments, before the contents of those log segments have been purged from LD's cache. Consequently, a discretionary cleaner can do its job efficiently because it does not have to read the blocks it wants to move into the storage area from disk first. Furthermore, if the cleaning is done between bursts of disk activity, the clients of LD will not experience a decrease in responsiveness of the disk.

On the other hand, a mandatory cleaner runs only when the head of the log has reached its tail. Under normal circumstances, the mandatory cleaner has little to do to move the tail of the log forward because the discretionary cleaner will have had enough opportunities to clean the data in the tail of the log. However, if the tail still contains live data, the mandatory cleaner reads these live blocks from disk and moves them to the storage area. Since it is likely that a mandatory cleaner has to read the live blocks in the tail of the log from disk first, the mandatory cleaner is less efficient than the discretionary cleaner.

Unfortunately, the performance measurements that we ran on LD did not include any idle time. Therefore, during our performance measurements, the discretionary cleaner cannot run during periods where the disk is idle to clean log segments. Consequently, all log segments written during our performance measurements will be purged from LD's cache before the discretionary cleaner has a chance to clean them. As mentioned before, if the discretionary cleaner has not cleaned the log segments, the mandatory cleaner will have to clean them, in order to create new empty log segments. Unfortunately, since the mandatory cleaner has to read the log segments from disk, the performance of LD will suffer significantly.

Therefore, to compensate for the lack of idle time in which the discretionary cleaner can do its job during our performance measurements, we have set up LD such that its discretionary cleaner will always deal with data blocks in the cache, before they are purged. We accomplished this by configuring the discretionary cleaner to start cleaning log segments whenever the maximum number of log segments that LD is allowed to cache has been reached. We call this version of the discretionary cleaner the **eager discretionary cleaner**.

Each time the eager discretionary cleaner runs, it cleans the four *oldest* log segments in LD's cache. Consequently, the number of uncleaned log segments is kept smaller than the amount of log segments that can be cached by LD. The result of this scheme is that the eager discretionary cleaner can do its job relatively efficiently because it does not have to read blocks from disk first. However, a more intelligent discretionary cleaner could be even more efficient as it could search all cached log segments, not just the four oldest, for groups of data blocks that can be written to the storage area efficiently. Grouping data blocks can reduce the amount and size of seeks necessary to clean log segments.

Notice that the mandatory cleaner in this scheme does not have much to do; the log segments at the tail of the log do not contain any live data blocks anymore because the eager discretionary cleaner has cleaned them all before. The mandatory cleaner can therefore move the tail of the log forward without copying any data to the storage area.

9.2.3 LD File System

LD provides only a low-level interface to disk storage. To evaluate how well LD's design is suited as a base for higher-level applications, such as a file system or a DBMS, we built an application on top of the prototype LD implementation. We implemented a file system on top of LD, called the **LD File System**, or simply **LDFS**. The reason to implement a file system and not a DBMS is that a DBMS contains much more functionality than a file system, such as query optimization and full transaction support. Consequently, a DBMS is much more difficult to implement. Furthermore, we want to evaluate LD and not the application on top of it, and therefore, we chose to keep the application as small as possible. Additionally, many different types of file systems are readily available, which allows us to compare the performance of LD's design to that of many other file system designs.

Similar to a standard file system, LDFS supports files and directories. The external interface of LDFS includes functions to manipulate files and directories. For example, this interface supports calls to create, delete, read, and write files and directories. Each

file and directory in LDFS has an i-node, which is similar to an i-node in FFS. In contrast to an i-node in FFS, however, an i-node in LDFS does not contain addresses of disk blocks since physical block management is done by LD underneath LDFS.

LDFS has the following properties:

- Each file is implemented by a disk file in LD for which physical clustering is requested.
- Each directory, which is just a table naming the files that are grouped within that directory, is implemented as the first disk file (i.e., diskfile_id 1) of a disk cluster in LD. Physical clustering is requested for this disk file.
- Files within one directory are implemented by disk files within the same disk cluster as the disk file that implements the directory. In other words, each directory is implemented in its own disk cluster. Physical clustering is requested for each disk cluster.
- The i-node of a file or directory is stored within the header of the disk file that implements that file or directory in LD (see Chapter 3).
- LDFS supports **immediate files** [Mullender and Tanenbaum, 1984], that is, files up to and including a size of 443 bytes are stored completely within the disk file header. 443 bytes is the size of a single disk block minus the size of an i-node in LDFS and minus one byte that is used by LD for internal purposes.
- Directories that are smaller than 444 bytes are also completely stored within the disk file header.
- Directory entries in a directory in LDFS are stored in a B-tree. The directory entries are alphabetically sorted on the file names. This method of storing allows LDFS to support directories containing thousands of files efficiently, because it makes searching within such large directories fast.
- Each operation on the file-system level, that is, each call to LDFS, such as a write, create or delete of a file or directory, is protected against system failures using an ARU. In other words, each operation has been made atomic.
- `ld_flush` is called only when LDFS is explicitly asked to do so. The use of ARUs protects the integrity of the files and directories on disk, and therefore, flushes do not have to be used so often, except when durability is absolutely required.

9.2.4 Other Utilities

Two additional programs were developed: `mkld` and `mkldfs`. These programs create an initial empty LD structure and an empty LDFS structure on a disk, respectively. The function of these programs is similar to the function of the programs `newfs` and `mkfs`, which are used to create an empty file system under FreeBSD and Linux, respectively.

9.2.5 Other File Systems

In our experiments, we compare LDFS on top of LD with a number of other file systems. These other files systems are summarized in Table 9.1. Below we will briefly discuss each of these file systems. A more thorough comparison is given in Chapter 10.

Table 9.1: Other file systems used in our experiments.

File system	Description
FFS	The standard Berkeley Fast File System
FFS-soft updates	The Berkeley Fast File System with soft updates
NetBSD-LFS	A log-structured file system.
Ext3	A journaling file system running on Linux.
JFS	IBM's journaling file system.
ReiserFS	Hans Reiser's journaling file system.
XFS	SGI's journaling file system.

Fast File System (FFS)

Berkeley's Fast File System (FFS) [McKusick et al., 1984] is the standard file system of FreeBSD [FreeBSD]. It uses synchronous metadata writes to try to minimize the damage to a file system caused by a crash. Additionally, it relies on the `fsck` program to fix any inconsistencies to the file system during recovery. In order to increase performance, FFS uses *cylinder groups*. Each cylinder group represents a physically contiguous amount of disk space. Each cylinder group contains i-nodes, a bitmap and data blocks. The bitmap of a cylinder group records which blocks of that cylinder group are used. Furthermore, to increase performance, the general rule in FFS is to store the blocks of files of a single directory within the same cylinder group together with the corresponding i-nodes. In our experiments, each cylinder group is 44 MB.

FFS with Soft Updates

Ganger et al. [2000] designed a method called *soft updates*, which avoids the need to order metadata writes with synchronous writes to guard the file system structure against inconsistencies due to crashes. Soft updates allows a file system to write metadata updates asynchronously to disk which improves its performance considerably.

The soft-updates technique works by keeping a dependency graph of all metadata updates. These dependencies indicate the order in which metadata updates must be written to disk in order to guarantee metadata integrity in case of a crash. When a dirty metadata block is about to be written to disk, the file system checks if that metadata block contains an update for which there are any unsatisfied dependencies. In other words, it checks whether there are updates in other metadata blocks that need to be written to disk before this metadata block. If so, the file system temporarily modifies the metadata block by

rolling back the metadata updates for which there were unsatisfied dependencies, before it writes the metadata block to disk. This temporary rollback accomplishes that metadata updates are propagated to disk in order. All these actions are done atomically, so that users are unaware of these actions. The technique of soft updates is further discussed in Chapter 10.

The FreeBSD system that we use also supports soft updates within FFS. In our experiments, we use the Fast File System both with and without soft updates. For simplicity, we will refer to FFS without soft updates as simply 'FFS' and to FFS with soft updates as 'Soft'.

NetBSD-LFS

The history of the *log-structured file system* or LFS starts in 1989 when Ousterhout and Douglis [1989] made a case to represent information on disk in the form of a circular append-only log. The first log-structured file system was Sprite-LFS [Rosenblum and Ousterhout, 1990, 1991, 1992; Rosenblum, 1992]. BSD-LFS [Seltzer et al., 1993] is another log-structured file system built for 4.4BSD. The LFS version we used in our experiments is NetBSD-LFS, which runs under the NetBSD operating system. NetBSD-LFS is directly related to BSD-LFS. In the experiments presented in this chapter, we will sometimes refer to NetBSD-LFS as simply 'LFS'.

Extended 3 File System

The standard file system used in the Linux operating system is the Extended 2 file system, or simply Ext2 [Ext2FS; Card et al., 1994; Beck et al., 1998]. More recently, Ext3 [Tweedie, 2000] has been developed by Stephen Tweedie, which extends Ext2 with a number of features, including *journaling*. Ext3 has been designed such that it is backward compatible to the Ext2 file system; users can mount an Ext3 file system as if it were an Ext2 file system without any problem, although the extra benefits of the Ext3 file system are not available then.

Ext3 supports different levels of journaling, which provide different levels of data integrity guarantees. In its default setting, Ext3 protects against metadata corruption, which avoids the need for prolonged file system recovery with `fsck`.

In our experiments, we tested Ext3 in two configurations, each with a different level of journaling. One version has Ext3's journaling level set to 'ordered', the default setting, which means that only metadata are being logged. The other version has Ext3's journaling level set to 'journal', which means that both user data and metadata are logged. This version of Ext3 is similar to LDFS which also logs both user data and metadata. In the remainder of this chapter, we will refer to both versions of Ext3 as 'Ext3-ordered' and 'Ext3-journal', respectively.

JFS

The Journaled File System (JFS) [JFS; Best, 2000] is a recoverable file system developed by IBM. JFS uses journaling, which allows it to guarantee file system consistency after a

crash. Unlike LD, however, JFS is not designed to log user data; JFS limits its recovery process to the file system structure only.

JFS was designed to be used in IBM enterprise servers. However, IBM has also released an open source version of JFS to the Linux community. We used this version of JFS for Linux to run our experiments.

ReiserFS

ReiserFS [Reiser] is a file system developed by Hans Reiser for Linux. ReiserFS is a journaling file system, which means that it can recover to a consistent state quickly after a crash. Another feature of ReiserFS is that it is based on balanced trees, which allows it to deal with small files efficiently.

XFS

XFS [Sweeney et al., 1996] is a file system developed at Silicon Graphics, Inc. (SGI). XFS was aimed at scalability with support for large files. It also includes (metadata) journaling technology to provide high reliability and rapid recovery. As part of SGI's open source project, an XFS version has been developed for Linux (see [Linux XFS]), which is the version we used in our experiments.

Test Results from NetBSD-LFS and JFS

In Section 9.5, we present and discuss the performance results from executing measurements on the above-mentioned file systems. However, even though we tested NetBSD-LFS and JFS, we have not included the results from these two systems in this chapter. The reason for not presenting the numbers for NetBSD-LFS is that NetBSD-LFS was too unstable and showed unexpected and unexplainable behavior during our tests. The cleaners of NetBSD-LFS, crashed in some of our tests, which made it impossible to test the file system fairly.

Furthermore, during most performance measurements NetBSD-LFS performed terribly. We analyzed this behavior and discovered that NetBSD-LFS writes excessive amounts of data, which was more than could normally be expected for the particular test. Whether this is normal behavior or some kind of bug, we do not know. We contacted the authors about this problem, but were not able to resolve the problem. Therefore, since NetBSD-LFS performance numbers were poor and probably not reliable, we decided not to present NetBSD-LFS's results.

The performance of JFS has been left out of our discussion because JFS was an average performer compared to the other tested file systems. In other words, JFS did not show particularly good nor particularly poor performance. Therefore, for the interest of clarity, we have left the results of JFS out of our discussions.

9.2.6 Hardware

Our experiments ran on a single PC. The specifications of this PC are given in Table 9.2. In order to be able to measure the disk activity of the file system being tested (e.g., LDFS

or FFS), the test machine contained two disks. The first disk held the operating system, which was not directly involved in the actual experiments, the second disk was the actual test disk which held the file system being tested.

Table 9.2: Specification of the test machine.

Component	**Test machine**
CPU	AMD K6 – 350 MHz
Memory	384 MB
Hard disk 1	ATA Seagate ST34321A, 4 GB
Hard disk 2	SCSI IBM Ultrastar 18ES DNES-309170, 9 GB
Operating Systems	FreeBSD 4.8-Release Linux (kernel version 2.4.19) NetBSD 1.6.1

Our test disk was the IBM Ultrastar 18ES DNES-309170, which is a 9 GB, Ultra-wide SCSI disk. The disk's main characteristics are summarized in Table 9.3. The tracks on the platters of this disk are divided into multiple zones (see our introduction to hard disks in Section 2.1.1 on page 18). The number of sectors per track ranges from 249 in the innermost zone up to 390 in the outermost zone. The sustained transfer rates for the innermost and outermost zone of this disk (12.0 MB/s and 18.8 MB/s, respectively) can be calculated using the formula given in Section 2.4.1 on page 33. Our performance measurements, which will be described in detail in Section 9.4, were all performed in the first couple of zones of the disk; therefore, the maximum theoretical throughput reachable in our measurements is 18.8 MB/s.

Table 9.3: Specification of the test disk.

Property	**Test disk**
Type	IBM Ultrastar 18ES
Total capacity (MB)	9,170
Bytes per sector	512
Sectors per track	249 – 390
Number of heads	5
Number of platters	3
Spindle speed (RPM)	7,200
Head switch time (ms)	1.6
Cylinder switch time (ms)	2.6
Average seek time (ms)	7.5
Sustained transfer rate (MB/s)	12.0 – 18.8

Figure 9.4 shows the utilization graph of the disk when transferring different amounts of data based on the maximum sustained transfer rate of 18.8 MB/s. The formula to calculate the bandwidth utilization of a disk was also given in Section 2.4.1. Chapter 5 described that LD currently uses direct segments of 256 KB and log segments of 1 MB. With such segments, LD should be able to reach a disk bandwidth utilization of 53% and 82%, respectively.

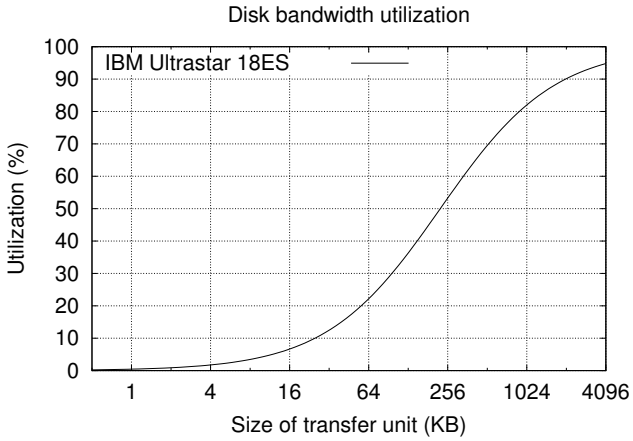


Figure 9.4: Disk bandwidth utilization of IBM Ultrastar 18ES DNES-309170, based on the maximum sustained transfer rate of 18.8 MB/s.

9.3 Source Code Statistics

One goal of LD is to improve the modularity of storage systems. LD takes care of low-level file management issues so that the clients on top of LD can concentrate on high-level file management issues. This separation makes the development and implementation of a storage system, such as a file system, easier. To evaluate how well LD has succeeded in improving modularity, we look at LDFS, a client on top of LD, and compare it to other existing file systems.

In this comparison, we compare the amount of source code for the tested file systems. Their approximate sizes are listed in Table 9.4. Unfortunately, the source codes of FFS with soft updates and FFS without soft updates are integrated into one source tree. To save time, we did not try to separate the two file systems, but only list the size of the source code of FFS with soft updates, which is the larger of the two. For LD we list two parts: the LD-library and LDFS. Even though we will not discuss the results of the file systems LFS and JFS in the remainder of this chapter, we have also included the statistics for their source code.

The last two columns in the table show the relative sizes of the source code with respect to the size of the combination (LD + LDFS) and LDFS, respectively. The table

Table 9.4: Size of the source code of the tested file systems.

File system	Source code			
	# of lines	size (KB)	% of size (LD+LDFS)	% of size LDFS
LD	28,983	610	70.8	242.1
LDFS	11,700	252	29.2	100
LD+LDFS	40,683	862	100	342.1
ReiserFS	20,475	662	76.8	262.7
JFS	31,819	799	92.7	317.1
XFS	131,060	3,627	420.8	1439.3
Ext3	14,309	402	46.6	159.5
Soft	19,774	577	66.9	229.0
LFS	17,232	494	57.3	196.0

shows that the combination of (LD + LDFS) is 862 KB, which is larger than all other tested file systems, except for XFS. This fact is not surprising as LD offers a great deal of functionality not normally found in file systems, such as ARUs and on-line reorganizations. Only XFS is larger with its exceptionally large size of over 3 MB. On the other extreme, Ext3, even with client data journaling, is only half the size of (LD + LDFS). The other file systems, ReiserFS, JFS, Soft, and LFS, are situated somewhere in between these two extremes.

However, compared to LDFS alone, all other file systems are larger. Since LDFS focuses only on file management issues, it is only 252 KB, which is roughly 1.5 to 3 times smaller than most other tested file systems. Compared to XFS, LDFS is 14 times smaller. This result supports our claim that file systems on top of LD can be smaller and thus simpler due to improved modularity.

In conclusion, even though LD alone is already as large as a complete file system such as ReiserFS or Soft, it has a number of advantages. First, LD offers functionality not readily found in other file systems, such as its ARU mechanism. Second, LD only needs to be implemented once. After that, its code can be reused for other storage systems. Moreover, since they can profit from LD's functionality, they can be smaller and simpler, as illustrated by LDFS.

9.4 Description of the Experiments

The synthetic performance measurements that we ran on the different file systems examined the performance of the file systems by creating, writing, reading and deleting files. Each measurement consisted of the following four test phases:

(1) **create** — create N empty files, twenty files per directory

(2) **write** — write X bytes into each of the files created in the create test-phase

(3) **read** — sequentially read the whole contents of each of the files written in the write test-phase

(4) **delete** — delete the files created in the create test-phase

The tests in these phases were executed in order and we call the execution of these four test phases a **test run**. A test run was repeated several times on each of the tested file systems. Each time we used a different combination for the numbers N and X. The combinations we used are listed below in Table 9.5. A sync call is issued only at the end of each test phase. Furthermore, in between two successive phases, the tested file system was unmounted so that the next phase starts with an empty cache.

Each test run for a particular file system (e.g., LD+LDFS, FFS, XFS, etc.) started with the same initial contents on the disk (i.e., an initial file system). The contents of this initial file system had been aged by an *aging process*, which will be described in Section 9.4.6. To keep the results of our performance measurements that ran on different file systems comparable, we used the same aging-process to create an initial aged file system for each of the tested file systems.

With these tests, we could measure the create, write, read, and delete performance of each file system separately. In addition, these tests also measured a file system's ability to cluster data blocks of files on disk: both intrafile clustering and interfile clustering. A file system's intrafile clustering was tested by reading whole files sequentially in the read test-phase. A file system's interfile clustering (i.e., clustering of files within one directory) was tested by reading the files per directory in the read test-phase. The better a file system was able to keep the data blocks of files clustered on disk, the better it performed in the read test-phase. Note that LDFS uses the clustering properties of disk files and disk clusters supported by LD to implement intrafile and interfile clustering.

We tested the clustering abilities of a file system under two extreme situations: a good-case and a bad-case scenario. In the good-case scenario, the files were read from the file system in the same order as they were written before. In this case, the files were written as a whole, from beginning to end, one directory at a time. With this write order, a file system is able to achieve good, if not the best, intrafile and interfile clustering. This scenario was followed in our write test-phase and the subsequent read test-phase, as describe above.

The bad-case scenario is described in Section 9.4.7. In this scenario, we read the entire contents of an aged file system, a directory at a time. This aged file system had been created by creating files randomly in directories (see Section 9.4.6), and therefore, measuring how fast the contents of this aged file system could be read gave an indication of the clustering ability of a file system under more difficult circumstances than in the good-case scenario.

Each file in the aged file system had been created by writing its entire contents from beginning to end in one write. An even-worse-case scenario would have been to write the data of a file in random order and to interleave the writes of all files. In other words, files would not be written sequentially, nor would the files be written one after the other, but they would be filled in pseudo-parallel. However, since in practice files in file systems are usually written as a whole and from beginning to end, we did not consider this extreme case and always wrote data to a file sequentially, from beginning to end.

Besides performing the test runs described above, we also performed other tests that were variations on these test runs. Instead of using an aged initial file system for each test run, we also performed the test runs on an empty initial file system. Furthermore, instead of creating the test files in the create test-phase in groups of twenty files per directory, we performed test runs in which all files were created in one large directory. The combination of an empty vs. aged initial file system and a single directory vs. multiple directories yielded four test cases in total.

However, we decided to present only the results of the test combination that we feel is the most realistic: the tests that were run on an aged initial file system and created test files in multiple directories. A file system running on an empty disk often behaves very differently when running on a nonempty disk. Therefore, since file systems are designed to hold data, it is more realistic to test a file system's performance on a nonempty file system. Moreover, in our measurements, up to 50,000 files are created in one test run. Therefore, if we would create so many files in one directory, the results of our test runs would depend on the ability of a file system to cope with large directories, which is a high-level file management issue. With our experiments, however, we want to evaluate LD's low-level file management support, and not particular implementation features of a client on top of LD, such as LDFS.

The various combinations for N (number of files) and X (size of files) used in our test runs are summarized in Table 9.5. The enumeration of file sizes is made up as follows:

(1) the series $A_n = 4^{(n+2)}$ for $n = 1, \ldots, 8$ (i.e., 64, 256, 1024, ...)

(2) the series $B_n = 10 \times 4^{(n+1)}$ for $n = 1, \ldots, 7$ (i.e., 160, 640, 2560, ...)

(3) the series $C_n = \sqrt{(A_n \times B_n)}$ in which the numbers are rounded to the nearest integer, for $n = 1, \ldots, 7$ (i.e., 101, 405, 1619, ...)

The series A_n leads to file sizes that are powers of 2. Since file systems are usually good at transferring amounts of data that are a multiple of the block size (typically 4 or 8 KB), such as the larger values of the A_n series, we also included other file sizes. The series B_n consists of file sizes that are 10 times a power of 2. This yields file sizes such as 10 KB, which does not match a multiple of a 4 or 8 KB block size.

The series C_n is derived from A_n and B_n. C_n yields file sizes that are somewhere in the middle of two corresponding values A_n and B_n. The union of all three series yields file sizes that are almost evenly spaced from 64 bytes to 1 MB. Each successor is approximately 1.6 times its predecessor.

9.4.1 Create Test-Phase

In the create test-phase a number of empty files were created. In order to prevent letting the overhead of directory lookups dominate the results of this test phase and the other test phases, we created files in several subdirectories. The files were created in the bottom layer of a three-level directory hierarchy. Each directory held a maximum of 20 files or subdirectories. Only the root level could hold more subdirectories; up to 125 in our measurements, which corresponded to the case where 50,000 files were created ($125 \times 20 \times 20 = 50,000$).

Table 9.5: Combinations for number and size of files used in our performance measurements.

File size (bytes)	# of files	Total size
64	50,000	3.05 MB
101	50,000	4.82 MB
160	50,000	7,63 MB
256	50,000	12.21 MB
405	50,000	19.31 MB
640	50,000	30.52 MB
1 KB = 1,024	50,000	48.83 MB
1,619	50,000	77.20 MB
2,560	50,000	122.07 MB
4 KB = 4,096	50,000	195.31 MB
6,476	20,000	123.52 MB
10 KB = 10,240	20,000	195.31 MB
16 KB = 16,384	20,000	312.50 MB
25,905	5,000	123.52 MB
40 KB = 40,960	5,000	195.31 MB
64 KB = 65,536	5,000	312.50 MB
103,622	2,000	197.64 MB
160 KB = 163,840	2,000	312.50 MB
256 KB = 262,144	500	125.00 MB
414,486	500	197.64 MB
640 KB = 655,360	500	312.50 MB
1 MB = 1,048,576	500	500.00 MB

Since the created files were empty (file size 0), this test focused on measuring a file system's metadata write performance. For example, FFS was expected to show poor performance because it used synchronous writes to write metadata. On the other hand, the logging file systems were expected to perform much better. The results of this test are discussed in Section 9.5.3.

9.4.2 Write Test-Phase

In the write test-phase, the files created in the create test-phase were filled with data. The files were written to in the order they were created. Since the files already existed (i.e., their metadata such as i-nodes had already been created), this test measured mostly a file system's ability to write user data. File systems that perform well would come close to the maximum transfer bandwidth of the disk.

The performance of LDFS and Ext3-journal were expected to be lower than the performance of other file systems for most file sizes. This difference was due to the fact that these two file systems used user data logging for supporting a higher level of user data integrity, which generally meant that user data were written twice. The results of this write test are discussed in Section 9.5.4.

9.4.3 Read Test-Phase

In the read test-phase, the data written into the files in the previous write test-phase were read. The files were read in the same order they were created. The performance of this test depended on how well the file systems clustered the files that were stored on disk: both intrafile and interfile clustering. The better the clustering, the closer the read performance would be to the maximum transfer bandwidth of the disk.

LDFS's performance in this test depended on how effective LD's disk layout was, which was explained in Chapter 8. Realizing an efficient disk layout was the responsibility of LD's cleaner and reorganizer processes. In the experiment, we tested three versions of LD. The versions differed in how much reorganizing was done after each of the test phases. We will explain the different versions of LD in Section 9.4.5. The results of the read test are discussed in Section 9.5.5.

9.4.4 Delete Test-Phase

The delete test-phase was the last phase of a test run. In this test, the files created in the create test-phase were deleted in the same order they were created. This test also measured the metadata performance of a file system. However, in contrast to the create test-phase, the results of this test phase also depended on the size of the deleted files, since large files have more metadata than small files (e.g., indirect blocks).

Similar to the create test-phase, the FFS file system was expected to show poor performance compared to logging file systems due to the synchronous writes in FFS. The results of the delete test are discussed in Section 9.5.6.

9.4.5 Cleaning and Reorganizing in LD

As we explained in Chapter 8, LD uses cleaner and reorganizer processes to cluster data blocks on disk. The cleaner processes are responsible for moving data blocks from the log area into the storage area. The task of the reorganizer processes is twofold. First, the reorganizer processes move data blocks within the storage area itself to achieve clustering of data blocks. Second, it resizes the CDS, CDL, and metadata areas, when the distribution of the disk space over these areas needs to change.

Cleaning and reorganizing is an integrated part of LD's design. Under the assumption that disk traffic is bursty, at least part of the cleaning and reorganizing can be done when the disk is idle, and consequently, their overhead will not or only partly be noticeable to LD's clients. However, we have not modeled idle time in our performance measurements because we send all commands sequentially to the disk as fast as possible. Therefore, with our performance measurements we cannot evaluate how the cleaner and reorganizer processes should use the idle time between bursts.

To make a more fair comparison between LD and other file systems, we decided to include at least part of the overhead of cleaning and reorganizing into the results of the performance measurements. Otherwise, LD would, for example, benefit from its collective writes into the log without also including the overhead of cleaning the data out of the log again.

Cleaning Overhead

To include the overhead of cleaning into the performance results of the performance measurements, we simply let the cleaner processes run during the disk trace generation phase of running an experiment, in which the disk trace was created. As mentioned before, the cleaner processes ran at predetermined moments. Additionally, during the disk trace generation phase, we forced the cleaner processes to clean the log area completely at the end of each test phase, that is, at the end of the create, write, read, and delete test-phases. In other words, the commands generated by the cleaner processes during a test run were also logged in the disk trace. Consequently, during the disk trace execution phase of the experiment in which we measured how fast the disk could execute the disk trace, the effects of cleaning were included in the results. In a real-life setting, however, we expect that cleaning is partly done in between bursts of disk activity, so that clients will mostly suffer only a small loss in performance due to cleaning. In our prototype, the eager discretionary cleaner did almost all of the cleaning by cleaning log segments while they were still cached in main memory (see Section 9.2.2).

Reorganizing Overhead

Unfortunately, we could not include the overhead of LD's reorganizer processes into the performance results. The reasons for this limitation are twofold. First, the prototype reorganizer processes are not yet able to run concurrently and independently of the rest of LD, which means that they cannot run while the test run is being performed. In our measurements, they ran at the end of a test phase, but that is not how they were designed to run. The goal is to let reorganizer processes run incrementally, that is, they perform

their task a small step at a time. This design allows running reorganizers to stop their job quickly if LD needs to serve new incoming client requests.

Second, the implementation of the reorganizer processes is currently not very efficient, which would unnecessarily lower LD's overall performance. For example, currently, the reorganizer processes start by scanning the entire Mapping from beginning to end to check the clustering of each disk file and if necessary restore the clustering. A more efficient implementation would be to keep some administration where disk changes have been made, and therefore, where some reorganization may be desirable. More work and research is required to make the reorganizers more efficient. For these two reasons, we decided to keep most of the activity of the reorganizers out of our tests.

Measuring the Effectiveness of Reorganizing

Fortunately, we could and did test the effectiveness of our disk layout scheme by letting the reorganizers run between test phases. However, we did not log their disk activity in the disk traces, and therefore, left their overhead out of our performance results. Nevertheless, we believe that the performance measurements are fair, because under the assumption that disk traffic is sufficiently bursty, reorganizing can mostly be done in between bursts. In Section 9.6 we provide some arguments that support this assumption.

To see the effectiveness of our data layout, we tested three versions of LD:

(1) **No reorganizing** — No reorganizer processes were run. In the performance results, we refer to this system as 'LD-no_reorg'.

(2) **Full reorganizing** — The reorganizer processes were run after the aging process and after the write test-phase. The reorganizer processes performed all the tasks listed in Section 8.7 on page 214. In the performance results, we refer to this system as 'LD-full'.

(3) **Limited reorganizing** — The reorganizer processes were run after the aging process and after the write test-phase. The reorganizer processes performed all the tasks listed in Section 8.7 on page 214, except for the task 'CDS slot reorder'. Instead of this task, a **CDS slot defragment** task was run which reordered all data blocks in a CDS slot such that all free space was at the end of the CDS slot.

    The difference between this task and the original CDS slot reorder task is that the data blocks do not have to be stored in logical block address order. Consequently, the CDS-slot-defragment task is less I/O intensive than the CDS-slot-reorder task. By combining the free space in a CDS slot, future write commands will benefit. Future read requests, however, will not benefit, as the data are not necessarily clustered in the CDS slots. In the performance results, we refer to this system as 'LD-limited'.

Since LD used variable sizes for the CDS, CDL, and metadata areas, LD needed to resize these areas in some write tests. A resize was necessary whenever a write test filled an area with so much data that it overflowed, which occurred in a few of our test runs. In the case of an overflow, reorganizer processes resized the areas as explained in Section 8.7 by executing the CDS-shrink, CDL-shrink, and metadata-resize tasks. After the resize had

completed, the test continued again. In a real-life situation, the resize is likely to occur during the time in which the disk is idle, because LD can anticipate on future disk space requirements of an area. Therefore, resizing will likely not interrupt LD's clients.

However, since our prototype reorganizer processes could not run in the background, we chose to include any disk accesses necessary during the process of resizing areas into our performance results. Unfortunately, since our reorganizers are not very efficient yet, the performance impact of resizing areas is clearly visible in the performance results.

As said before, we will use the terms LD-no_reorg, LD-limited, and LD-full in the remainder of this chapter to refer to the combination of LDFS and LD without reorganization, or LDFS and LD with limited reorganization, or LDFS and LD with full reorganization, respectively.

Measuring Sustained Performance

As mentioned before, LD's cleaners and reorganizers are designed to run mostly during idle time so that clients hardly notice the overhead of their work. How bursty disk traffic needs to be (i.e., how much idle time is available) for allowing the cleaners and reorganizers to do their job in between bursts, depends on the amount of work that needs to be done and on the efficiency of the cleaner and reorganizers. The more efficient (i.e., faster) cleaners and reorganizers are, the less idle time is necessary for them to do their work. Currently, in the prototype of LD, the cleaners and reorganizers are not very intelligent. For example, the eager discretionary cleaner simply cleans the oldest four log segments from LD's cache at a time. In Section 9.6, we briefly look at how much idle time is necessary for LD to perform its reorganizations.

Recall that there is no idle time in our experiments. Therefore, since we included all the cleaning overhead in the performance results, the numbers we present in this chapter for LD actually correspond to a bad-case scenario. This scenario represents LD's sustained performance under the restriction that there is no idle time to reorganize. In fact, this scenario represents a lower bound on LD's performance. Under normal circumstances, at least some of the cleaning overhead would be performed during idle time, and consequently, LD's performance would be higher than presented here. Moreover, if the experiments included idle time, the disk could (partly) be reorganized during the experiments, which could also increase LD's performance in the remainder of the experiments. Last, a more efficient and intelligent cleaner may also raise LD's performance over the numbers given in this chapter.

9.4.6 Aging the File System

In order to make the synthetic performance measurements more realistic, we ran the measurements on a nonempty file system. Furthermore, before running the measurement, we aged the file system in order to test a file system's ability to cope with aging and the resulting fragmentation of free space. The effects of aging have been discussed in Chapter 8.

We used an aging-process to create an aged file system that contained 41,358 files in 2,196 subdirectories underneath a root directory. The average number of files per directory was 18.8 and the maximum number of files per directory was 43. The combined size of

the files was 954 MB and the average file size was 23.6 KB. The exact create and delete commands that were used to create this aged file system were saved. These commands were then reused to create the same aged file system for each of the tested file systems. By using the same commands, we created equivalent initial aged file systems for LD+LDFS, FFS, ReiserFS, etc, so that we could compare the performance results from performing test runs on different file systems with each other.

We used the following algorithm to create an aged file system:

(1) Create an empty 2 GB file system.

(2) Fill the file system with files and directories until it contains 1 GB of data.

(3) Age the file system by repeatedly creating new files and deleting existing files.

The first step, creating the empty file system, was done by calling the appropriate file system utility of the tested file system (e.g., `newfs` for FFS, or `mkfs` for Ext3). The second step filled the file system with files. The newly created files were created in subdirectories of a two-level directory hierarchy. During this step, enough subdirectories were created to keep the average number of files within a directory at 20. Each newly created file was created at random in one of the existing subdirectories. The size of newly created files was chosen using a distribution that had been obtained in a survey on file systems held by Gordon Irlam with the help of the Internet community in 1993 [Irlam, 1993]. The survey covered over 12 million files from 1,050 file systems taking up 259 GB of disk space. The distribution of file sizes is summarized in Table 9.6.

Note that the table summarizes the file sizes by dividing them into buckets showing a maximum size. The aging process determined a file size for a new file by first choosing a bucket, using the distribution as given in the table. Next, the actual file size of the newly created file was randomly chosen within that bucket. Since our file system was only 2 GB large, we ignored the last bucket.

The third and last step of creating an aged file system was done after the empty file system had been filled up to 1 GB (i.e., the file system is 50% used) in the second step. This last step consisted of actually aging the file system by creating and deleting files. This process was done as follows. The aging process simply kept on going creating files using the distribution of Table 9.6. However, whenever a new file was to be created that would have brought the total size of the file system over 1 GB, the aging process first deleted existing files until the new file could be created without bringing the total size over 1 GB. The files that were deleted were chosen at random, which kept the distribution of file sizes on the aged file system close to the distribution as found by Gordon Irlam. This final step stopped as soon as at least 100,000 files had been deleted. The new file that had caused the 100,000-th file to be deleted was the last file that was written to disk. Note, in order to ensure that after this step the resulting file system had been sufficiently aged (i.e., fragmented) for our performance measurements, we checked that at the end of this last step there were no deletions of very large (i.e., several hundreds of MB) files. Such deletions could have resulted in large consecutive free block ranges, which could have subsequently been re-used to store many new, smaller files sequentially, which would have resulted in a less fragmented file system.

Table 9.6: Distribution of file sizes and the amount of space they consume. From a survey on 1,050 file systems held by Gordon Irlam to study UNIX file sizes [Irlam, 1993].

File size (max. bytes)	#Files	%Files	%Files cumm.	Disk space (MB)	%Space	%Space cumm.
0	147479	1.2	1.2	0.0	0.0	0.0
1	3288	0.0	1.2	0.0	0.0	0.0
2	5740	0.0	1.3	0.0	0.0	0.0
4	10234	0.1	1.4	0.0	0.0	0.0
8	21217	0.2	1.5	0.1	0.0	0.0
16	67144	0.6	2.1	0.9	0.0	0.0
32	231970	1.9	4.0	5.8	0.0	0.0
64	282079	2.3	6.3	14.3	0.0	0.0
128	278731	2.3	8.6	26.1	0.0	0.0
256	512897	4.2	12.9	95.1	0.0	0.1
512	1284617	10.6	23.5	566.7	0.2	0.3
1024	1808526	14.9	38.4	1442.8	0.6	0.8
2048	2397908	19.8	58.1	3554.1	1.4	2.2
4096	1717869	14.2	72.3	4966.8	1.9	4.1
8192	1144688	9.4	81.7	6646.6	2.6	6.7
16384	865126	7.1	88.9	10114.5	3.9	10.6
32768	574651	4.7	93.6	13420.4	5.2	15.8
65536	348280	2.9	96.5	16162.6	6.2	22.0
131072	194864	1.6	98.1	18079.7	7.0	29.0
262144	112967	0.9	99.0	21055.8	8.1	37.1
524288	58644	0.5	99.5	21523.9	8.3	45.4
1048576	32286	0.3	99.8	23652.5	9.1	54.5
2097152	16140	0.1	99.9	23230.4	9.0	63.5
4194304	7221	0.1	100.0	20850.3	8.0	71.5
8388608	2475	0.0	100.0	14042.0	5.4	77.0
16777216	991	0.0	100.0	11378.8	4.4	81.3
33554432	479	0.0	100.0	11456.1	4.4	85.8
67108864	258	0.0	100.0	12555.9	4.8	90.6
134217728	61	0.0	100.0	5633.3	2.2	92.8
268435456	29	0.0	100.0	5649.2	2.2	95.0
536870912	12	0.0	100.0	4419.1	1.7	96.7
1073741824	7	0.0	100.0	5004.5	1.9	98.6
2147483647	3	0.0	100.0	3620.8	1.4	100.0

9.4.7 Miscellaneous Experiments

Besides the experiments described above in Sections 9.4.1 through 9.4.4, we performed three additional experiments. The first experiment examined LD's peak write performance. The second experiment was another read performance test. The last experiment measured the recovery time of the file systems after forcing a crash. Below we will describe each experiment in turn.

LD's Peak Write Performance Experiment

In this experiment, we examined the write performance of LD under the assumption that it could run all its cleaner and reorganizer processes in the background. This situation is realistic, at least, if disk traffic is sufficiently bursty so that LD's cleaners and reorganizers only have to run when disk traffic is low. We can measure LD's write performance in such situations experimentally by turning off cleaning altogether and writing a small amount of data only, so that these data fit within the log on disk. In fact, this experiment measures the peak write performance LD can achieve in real-life situations.

This experiment consisted of running the create and write test-phases on an aged file system that had been fully reorganized. However, the number of files created during the write test-phase was changed such that the total amount of data written into the log during the write test-phase fitted within the log area on disk. In this experiment, no more than 128 MB was written into the log during the write test-phase. The results of this experiment are presented in Section 9.5.1.

Reading an Aged File System

The previous read experiment described in Section 9.4.3 measured the read performance of the file systems while reading files of one particular size only. Furthermore, these files were read in the order they were created, that is, one directory at a time. To measure the read performance while reading files of varying sizes that were created in a more-or-less random order, we performed an additional experiment. This experiment consisted of reading the entire contents of the file system after our aging process, as described in Section 9.4.6, one directory at a time.

To make a fair comparison, the files in the aged file system were all read in the same order for each file system. This order was determined by executing a UNIX `find` command on the aged FFS file system. This command listed all files within a directory of the aged file system, one directory at a time. The files in a directory were listed in the order they were stored within the directory of the aged FFS file system. This order may not always have been the most optimal order for the other file systems to read the files in a directory. It certainly was not the optimal order for LD, as the diskfile_id's used for the files in a directory were chosen independent of this order. Consequently, the disk file's meta information was stored in LD's Mapping in a different order.

Furthermore, since the aging process created the files in a random order and this experiment read the files one directory at a time, this experiment tested a file system's ability to keep the files clustered per directory during the aging process. This read test perhaps

simulated a more realistic environment than the synthetic performance measurements described in Sections 9.4.1 through 9.4.4. In those performance measurements, files were created, written, read, and deleted in the same order, namely, one directory at a time. An example where such conditions more or less occur is when a user unpacks a zip-archive or tar-archive containing a source-code tree in a directory, subsequently compiles the source code, and finally removes the source code again. However, the read test described in this section, simulates the use of a file system that has aged over time. The results of this read test are discussed in Section 9.5.2.

Crash Recovery Experiment

LD and Ext3-journal offer more extensive data integrity guarantees than the other file systems tested. In this last experiment, we measured whether this improved data integrity had much impact on how fast LD and Ext3-journal could recover from a file system crash compared to other file systems. The crash recovery test consisted of running the create and write test-phases in which 50,000 4 KB files were created and written on an aged file system. However, after some thirty seconds into the write test-phase, we pressed the reset button of the test computer to simulate a crash. During recovery, we traced the disk accesses performed during the mount and, if required, included the accesses performed during the running of an `fsck` program on the crashed file system before the mount. The results of this experiment are discussed in Section 9.5.7.

9.4.8 Command Reordering within the Disk

By using disk traces in our test method, we are able to look only at disk accesses during the performance measurements. Unfortunately, there are two complications when we time the results of executing the disk traces on our test machine in the disk trace execution phase of an experiment. These complications are caused by two optimizations that the hard disk and the SCSI-bus offer: *write caching* and *tagged command queuing*, respectively. Both optimizations can change the execution order of the disk commands, and can therefore, when used thoughtlessly, counteract any data integrity guarantees offered by a file system.

Write Caching

All recent hard disks have an internal cache, with sizes up to several MBs. For example, our IBM test disk has a 2 MB internal cache, which is used as a read cache as well as a write cache, if write caching has been enabled. With write caching enabled, the disk will acknowledge a write command as completed to the host computer, as soon as the data that are to be written have been stored in the cache of the disk, that is, before they are really stored on disk. The disk's firmware will write the data to the actual disk medium at a later point in time.

The advantage of write caching is that the firmware of the disk can write the data from its write cache to the actual disk platters in an efficient order. For example, this order could be chosen to minimize the seek distance. Unfortunately, a disadvantage of write caching is that a power failure will lead to the loss of the contents of the disk's internal write cache, and therefore, write caching can compromise the integrity of the data on disk.

However, potentially more dangerous to the integrity of data on disk is the fact that write caching may effectively change the execution order of write commands, since the disk may decide the order in which to write cached data to the actual disk. Recall that synchronous writes are used to enforce that updates — usually only metadata updates — are propagated to disk in a specific order. In other words, the host computer waits until the disk acknowledges that a write has completed, before sending the next write command to the disk. By enforcing a specific write order, the possible inconsistencies that can occur in the file system on disk due to a crash are limited to a number of known cases, and those cases can easily be recognized and fixed during recovery. With write caching enabled, however, the disk can still change the order in which updates are propagated to disk even though the disk acknowledges the write commands in order. Consequently, during recovery, the inconsistencies in the file system may not be easily fixable.

Tagged Command Queuing

SCSI's tagged command queuing (see e.g., [Schmidt, 1995]) is a feature of the SCSI standard that allows multiple commands to be outstanding to the disk at a time. Without this feature, a command can only be sent to the disk after the previous one has been completed, which slows down performance. With tagged command queuing, however, the host computer can send a request to the disk while the disk is still executing the previous request, which allows the time of sending a request to overlap with the time needed to execute a previous request. Moreover, the disk itself usually chooses the order in which it executes the outstanding commands.

Each command has been 'tagged' so that when sending a reply to the host computer, the disk can signal the host to which command the reply applies. Hence the name 'tagged command queuing'. These tags are called *simple tags*. Obviously, the reordering can also endanger the integrity of the data on disk when used carelessly. It is, however, also possible to use tagged command queuing without the danger of reordering commands, by tagging the commands with special *ordered tags*. These ordered tags prohibit the disk from reordering the commands, but multiple commands can still be outstanding to the disk at a time. It is also possible to mix commands with simple and ordered tags. The effect of an ordered tag is that all commands sent before the command with the ordered tag will be executed before it, and all commands sent to the disk after the command with the ordered tag will be executed after it.

Command Reordering: Off or On?

In this subsection, we look at the consequences of enabling write caching and tagged command queuing with simple tags. Above we argued that command reordering endangers the data integrity guarantees made by file systems. However, in most operating systems, the default setting is that both optimizations are enabled (i.e., write caching and tagged command queuing with simple tags are used) to achieve better performance, and therefore, they ignore the consequences of potential data integrity inconsistencies!

It is strange to realize that much effort and research has been spent on developing and integrating new techniques, such as logging and soft updates, into file systems to guarantee the integrity of (meta)data, and yet, operating systems such as Linux and FreeBSD by

default turn on write caching and tagged command queuing with simple tags. This default setting, in fact, voids the guarantees made by synchronous writes as well as these new techniques concerning data integrity. This issue once more illustrates the importance of clearly defining semantics with respect to (meta)data integrity and implementing them correctly.

Since we advocate that (meta)data integrity guarantees are important, it seems logical that we advocate to disable command reordering. In other words, we advocate to turn off write caching and to use ordered tags. This measure, however, unnecessarily lowers a file system's performance because it is too strict. For example, since most file systems guarantee only metadata integrity, they enforce a specific write ordering only on metadata writes. Consequently, such file systems can profit from changing the write order on other write commands without endangering the guaranteed metadata integrity. In short, not all commands need to be executed in order, only the ones that are important to guarantee (meta)data integrity. Fortunately, it is possible to use both optimizations and still guarantee (meta)data integrity .

For example, let us assume that a file system uses synchronous writes to guarantee the order in which data blocks are written to disk to maintain file system consistency in case of system failures. In this case, both write caching and tagged command queuing with simple tags can safely be used if each synchronous write would use the *ordered tag* and the special SYNCHRONIZE_CACHE SCSI command would be sent at the end of each synchronous write. Said SCSI command forces the entire contents of the disk's cache to disk. With these measures, a file system's consistency can be guaranteed even if both optimizations are enabled. Unfortunately, these measures are not generally used in file systems. For example, we did not find it when we examined the source code of FFS in FreeBSD.

A general solution to guarantee command ordering for important data blocks is to use *barrier* points. A barrier has the property that all write commands sent to the disk *before* a barrier was issued must all have been executed and their data must have reached the disk, before any write command sent *after* the barrier is executed by the disk. A barrier point can be implemented, for example, by using the *ordered tag* and the SYNCHRONIZE_CACHE SCSI command. (Another method is to temporarily stop sending new commands to the disk until all currently outstanding commands have been acknowledged by the disk, and then send the SYNCHRONIZE_CACHE SCSI command.)

A file system could use these barriers at specific moments when writing data to ensure the correct ordering of write commands. For example, LD should issue a barrier *before* and *after* writing a log segment. The barrier before the write of the log segment is necessary to ensure that the data blocks, such as blocks in direct segments (see Chapter 5), reach the disk before the log tuples in the log segment describing those data blocks reach the disk. The barrier after the write log segment is necessary to ensure that disk space freed by commands whose log tuples are in the log segment can be safely reused by future commands.

If file systems used these barriers, both optimizations of write caching and tagged command queuing with simple tags could continuously be left on, which provides improved disk performance to the file systems. However, without these barriers, the only way to uphold the guarantees that file systems make with respect to (meta)data integrity

is to disable command reordering.

Presenting the Performance Results

The question answered in this subsection is whether we should enable write caching and tagged command queuing with simple tags when we run the disk traces on the test machine during the disk trace execution phase of our measurements. If the barriers, which we presented previously, were used in the tested file systems and were present in the disk traces, we could have turned the optimizations on, and still uphold the (meta)data integrity guarantees offered by the file systems. These barrier points, however, are currently missing from our disk traces. Adding these barriers to the disk traces of LD is relatively simple as we know where the barrier points need to be placed. Unfortunately, manually adding these barriers to the disk traces of the other file systems would require us to analyze their source code to find the correct barrier points, which would simply be too time consuming.

Furthermore, if we turned both optimizations on during the disk trace execution phase, the performance numbers would then potentially be too optimistic. When we run a disk trace, we send all commands in the trace to the disk as fast as it can take them, completely ignoring the amount of time originally separating the two subsequent commands during the actual running of the test that generated this disk trace. Consequently, during the running of a disk trace, the disk firmware can potentially reorder two commands and hence gain performance when in reality, the disk firmware could not have reordered those commands because in real time the latter command would simply not have been issued yet. For example, a write command that is sent after a synchronous write command would, normally, not appear in the disk's command queue until after the disk has executed the previous synchronous write command. However, when running the disk trace, the second write command would be sent to the disk immediately after sending the first.

On the other hand, if we choose the setting that upholds the file system's (meta)data integrity guarantees by turning off both optimizations (i.e., the write cache is disabled and the commands in the disk trace are tagged with ordered tags), the measurements would, unfortunately, yield performance results that are too conservative. These results are too conservative because of the following two reasons:

(1) Not all commands need to be executed in order, only the ones that are important for guaranteeing (meta)data integrity (see above).

(2) Due to hardware restrictions, our test platform is unable to execute two consecutive write commands 'back-to-back' with the write cache disabled. Two write commands are consecutive if the second write command writes a range of data blocks to disk at a position that starts immediately after the end of the range of data blocks written by the first command. For example, if the first command writes blocks 100 – 129 and the second command writes blocks 130 – 150, these two commands are consecutive. Writing back-to-back means that the disk can execute the second command of the two consecutive write commands immediately after executing the first command, without losing a full rotation. Unfortunately, our test disk cannot start executing the second command fast enough. Consequently, the starting position of the second write command has rotated past the disk head before the disk is ready to

execute the second command, forcing the disk to wait a full rotation (8.3 ms on a 7,200 RPM disk).

The second reason, the inability to write back-to-back, is the consequence of a hardware limitation. Unfortunately, this limitation has a rather big impact on our performance numbers. At first sight, a work-around for the disk's inability to write back-to-back is to alter the disk trace by coalescing two or more consecutive write commands into one large sequential write command before executing the disk trace. Unfortunately, the kernel limits the disk trace driver to issue read and write commands to the disk to a maximum of 64 KB. In other words, the disk trace driver must split up all commands in the disk trace that involve reading or writing more than 64 KB into two or more consecutive commands, which will lead to lost rotations because the disk cannot write back-to-back. Therefore, even altering the disk traces will not completely avoid lost rotations. Turning write caching on and using tagged command queuing with simple tags does enable the disk to write commands back-to-back because with write caching enabled, the disk can first collect all data into its cache and write the data to disk later.

In conclusion, turning the optimizations on — in some operating systems the default setting — yields an upper bound, and turning the optimizations off — which upholds (meta)data integrity guarantees —- yields a lower bound on the performance we want to measure. Therefore, we have chosen to produce both sets of performance numbers. One set is the result from running the disk traces on a disk *with* write caching and with *simple* tags. The other is from running the disk traces on a disk *without* write caching and with *ordered* tags. The first set of numbers more or less corresponds to the situation where file systems would use *barriers* to enforce a particular execution order for commands that are important for (meta)data integrity so that other commands can benefit from write caching and simple tags without endangering the overall (meta)data integrity guarantees. The second, lower, set of numbers corresponds to the situation where barriers are not used, but (meta)data integrity must still be enforced.

Fortunately, the two sets of performance numbers resulting from our measurements are very similar in certain aspects. If the performance results in one set reveal that a particular file system behaves well or poorly with respect to other systems in one particular test, the other set of performance results reveals the same behavior for that same test, with only a few exceptions. A major difference is that the absolute numbers (i.e., how fast it can create, write, read, or delete a file), will be higher in one set. In other words, file systems that perform well in one performance set also perform well in the other; similarly, poorly performing file systems perform poorly in both sets. Based on this observation, it seems reasonable to use these results to determine whether LD provides competitive performance to other file systems, even though the performance results present an upper and lower bound.

Since the two sets of performance results show many similarities, we have split the presentation of both sets for clarity. In this chapter, we present only the performance results with the optimizations turned on (i.e., write caching enabled and tagged command queuing with simple tags) in Section 9.5. These numbers show the performance that the tested file systems should be able to achieve, if barriers had been used correctly. The performance results with the optimizations turned off (i.e., write caching disabled and tagged command queuing with ordered tags) are included in Appendix A.

9.5 Performance Results I

As explained above, we present two sets of performance numbers. This section presents the performance results in which we enable write caching and use tagged command queuing with simple tags in the disk when we run the disk traces through our trace driver. Appendix A presents the second set of performance numbers in which write caching has been disabled and tagged command queuing uses ordered tags.

This section is structured as follows. First, we present the results of LD's peak write performance experiment and the experiment in which the contents of an aged file system are read, as presented in Section 9.4.7. These results will quickly give an impression of LD's performance. Subsequently, the performance results of each of the four test phases, as described in Sections 9.4.1 through 9.4.4, are presented in turn. Finally, we present the result of the crash experiment, which was described at the end of Section 9.4.7.

All the performance numbers are the average of running the disk traces three times through our trace driver. In over 99% of the cases, the variations in the performance numbers measured in these three runs were within 5%. Deviations up to 10% were also present, but in these cases the total absolute time was mostly in the order of only one second.

9.5.1 LD's Peak Write Performance Experiment

In this experiment, we examined the write performance when LD is able to run all its cleaner and reorganizer processes in the background. Figure 9.5 shows the result of this performance measurement.

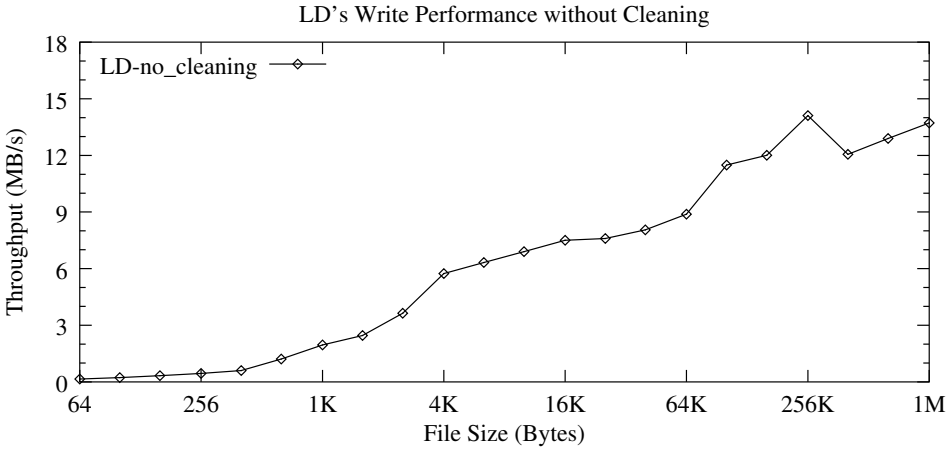


Figure 9.5: Results of the write test-phase on an aged file system. The line is from LD with all cleaning turned off. The disk had write-caching enabled and used tagged command queuing with simple tags.

As expected, the peak performance of LD without cleaning rises as the file size gets bigger. The peak write performance is achieved at 256 KB files, where all data are written via direct segments directly into the storage area, bypassing the log. Those direct segments can be written efficiently because LD is able to find large consecutive ranges of free space in the CDL area, that is, the part of the storage area into which direct segments are written. Unfortunately, the measurement shows an anomaly: the peak performance reaches only a little over 14 MB/s. In another test, we will see that LD is able to reach over 16 MB/s (see Section 9.5.4).

After 256 KB, the graph drops slightly because the tails of files whose size are not a multiple of the size of a direct segment are written in the log area, while the rest of those files is written as direct segments. As a consequence, the in-core log segment fills up quicker with data than if only files of exactly 256 KB would be written, because then only log tuples would be written in the in-core log segment. Therefore, for files that are larger than 256 KB, but not an exact multiple of 256 KB, more log segments are written, which means that more seeks are required between the log area and the CDL area where direct segments are written, which lowers performance a little. The performance impact of the seek for files larger than 256 KB, gets smaller as files get bigger, because the relative size of the tail gets smaller compared to the rest of the file whose data can be written efficiently in direct segments.

The line in this graph represents an upper boundary for the write performance of LD. If disk traffic is sufficiently bursty so that LD can perform all cleaning in between bursts, LD will reach its peak performance as shown by this line. If there is no time for cleaning, LD will achieve a performance below this line. A lower bound for LD's write performance is given in Section 9.5.4. In conclusion, LD can achieve good to very good write performance if LD can perform most cleaning in between bursts of disk traffic.

9.5.2 Reading an Aged File System

This experiment consisted of reading the entire contents of the file system after our aging process, as described in Section 9.4.6, one directory at a time. The performance results of this experiment are listed in Table 9.7. The last column shows the relative read performance of the file systems compared to the read performance of LD-full. A graphic representation of the results is presented above the table.

From the table, it is clear that LD realizes very good read performance. With full reorganization, LD considerably outperforms the other tested file systems by a factor of two to three. With limited reorganization, LD still outperforms the others up to almost a factor of two. Moreover, even without reorganization, LD already has competitive read performance; only XFS shows a marginally better read performance. Analysis of the disk traces showed that LD without reorganization performs the most seeks of all the tested file systems. The probable reason why LD-no_reorg still outperforms the other file systems except for XFS is that, in contrast to the other file systems, more than half of LD's seeks are so small that the read-ahead cache of the disk can often satisfy the request.

In LD-limited and LD-full not only the total number of seeks drops compared to LD-no_reorg, which is to be expected, but also the percentage of small seeks rises slightly, which results in even better read performance. The total number of seeks (including small

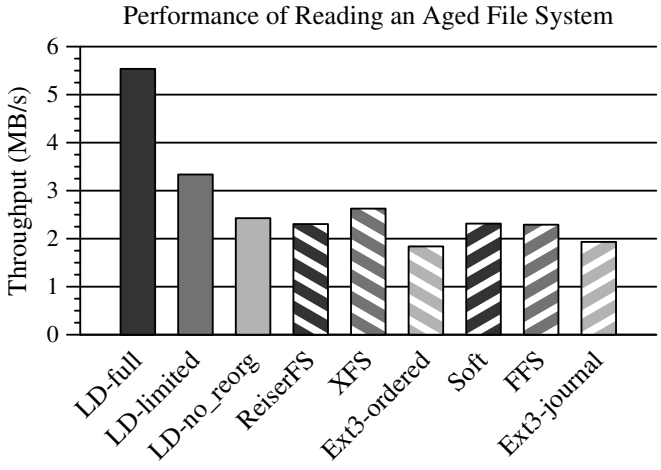


Table 9.7: Results of reading the aged file system. The disk had write-caching enabled and used tagged command queuing with simple tags.

File system	Read throughput (MB/sec)	% of perf. LD-full
LD-full	5.54	100
LD-limited	3.34	60.3
LD-no_reorg	2.43	43.9
ReiserFS	2.30	41.5
XFS	2.63	47.5
Ext3-ordered	1.84	33.2
Soft	2.32	41.9
FFS	2.29	41.3
Ext3-journal	1.93	34.8

seeks) performed by LD-full is already 1.5 to 2.5 times smaller than the total number of seeks of the other file systems. In fact, LD with full reorganization reaches an average read request size of 23.1 KB, which is very close to the average file size of 23.6 KB in the aged file system (see Section 9.4.6), whereas the closest competitor XFS reaches only 15.3 KB.

The reason why so many seeks of LD are small is due to the way LD has structured the CDS area. Recall from Chapter 8 that the CDS area is divided into CDS slots, which are each 2.5 MB in size. LD tries to keep the disk files of a single disk cluster in one CDS slot, which in our experiment means that files of a single directory are stored close to each other. Since CDS slots are only 2.5 MB, LD has relatively many CDS slots in a 2 GB file system, which was used in our tests. Therefore, since the aging process created files in 2,196 directories, relatively few directories have to share a single CDS slot. Consequently, even though the aging process created files randomly, the files of a single directory will not only have good intrafile clustering, as the blocks of the files are all within the same slot (if there is room in the slot), but will also have good interfile clustering, as not many files of other directories can interfere.

In contrast, even though a file system such as FFS also tries to support interfile clustering by storing files of a single directory in the same cylinder group, this approach here is less effective than LD's approach. The reason that cylinder groups do not perform as well is because cylinder groups are much larger than LD's CDS slots. In our test, each cylinder group was 44 MB, and therefore, FFS had fewer cylinder groups than LD had CDS slots, which meant that relatively many directories had to share a single cylinder group. Consequently, since the aging process created files randomly, the interfile clustering suffered. Our guess is that the other tested file systems likewise could not keep the interfile clustering at a good level because the aging system created files randomly in directories. For instance, Ext3-ordered and Ext3-journal use large *block groups* which are similar to cylinder groups in FFS.

9.5.3 Create Test-Phase

In the create test-phase, we measure how fast file systems can create empty files in an already aged file system. The number of files created during test runs ranged from 500 files to 50,000 files (see Section 9.4). The performance of a particular file system in this create experiment depends on how many files are created in a test run. However, we present only the performance results of the test runs that created 50,000 files, as we consider this experiment measures a file system's sustained file creation performance the best. Table 9.8 summarizes the create performance each file system achieved creating 50,000 files. The last column in the table shows a file system's create performance relative to the create performance of LD-full. Above the table, a graphic representation of the create performance results is shown.

The results of the file systems have been listed in three groups. The first group shows the results of the three versions of LD: LD with full reorganization, LD with limited reorganization, and LD without reorganization. The next group shows the metadata logging file systems ReiserFS, XFS, and Ext3 with the level of logging set to 'ordered'. The last group shows the remaining tested file systems: FFS with soft updates, FFS without

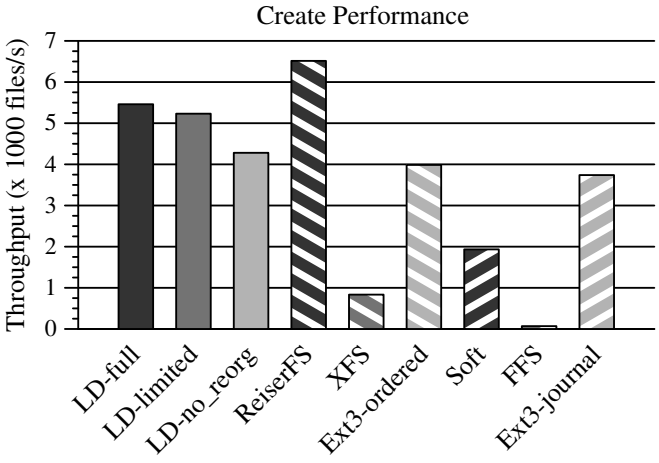


Table 9.8: Create Performance. Results of the create test-phase for 50,000 files. The disk had write-caching enabled and used tagged command queuing with simple tags.

File system	Create throughput (files/sec)	% of perf. LD-full
LD-full	5,461	100
LD-limited	5,231	95.8
LD-no_reorg	4,281	78.4
ReiserFS	6,514	119.3
XFS	834	15.3
Ext3-ordered	3,986	73.0
Soft	1,934	35.4
FFS	69	1.3
Ext3-journal	3,740	68.5

soft updates, and Ext3 with the level of logging set to 'journal'. In the remainder of this chapter, we will use this grouping to structure the performance results, which makes them easier to discuss.

The level of reorganizing has an influence on LD's create performance. The reason is that reorganizing the data after aging the initial file system clusters the client's data on disk, which enables LD to store its Mapping compactly in the metadata area (see the compression methods of the Mapping in Section 6.2.3). Consequently, LD has to read fewer disk blocks from the metadata area during the create test-phase. Especially, since in the current implementation of LD metadata is not clustered in the metadata area, a small Mapping can save a lot of random seeks, which are expensive.

ReiserFS has the best create performance of all tested file systems. Both versions of Ext3 perform a little under the performance of LD-no_reorg. XFS shows poor performance because it writes a lot of data, which can reach over 200 MB when creating 50,000 empty files. The worst performer is FFS, whose synchronous updates cause each single update to a metadata block to be forced to disk. This results in FFS writing almost 500 MB worth of data when creating 50,000 empty files. The above performance results show that synchronous writes are really bad for performance.

In conclusion, even though LD with full reorganization does not show the best create performance, it does show very good performance which is certainly competitive to other file systems.

9.5.4 Write Test-Phase

The results of the write performance test is shown in Figure 9.6. For clarity, the results are divided into three pairs of graphs. For ease of reference, the results of LD with full reorganization has been repeated in all three pairs of graphs. In each pair of graphs, the left graph shows an enlargement of the right graph for file sizes up to 1 KB. Notice that the scales for the left and right graph are different.

The first pair of graphs shows the performance results of LD-no_reorg, LD-limited, and LD-full. In addition, we added the graph of LD's peak write performance measurement, which is marked as LD-no_cleaning. Comparing the performance results of LD-no_reorg, LD-limited, and LD-full, one can see that reorganizing is very beneficial for the write performance. The difference between LD-limited and LD-full is almost negligible, which is not surprising as both types of reorganizing generate the same amount of consecutive free space. The lack of sufficient consecutive free space is what causes LD-no_reorg to have a lower performance.

The performance for files up to and including 405 bytes remains the same for all four lines in the graph. Such files are always written twice, once in the log for recovery purposes and once in the metadata area, because they are stored in file headers in the Mapping. For files larger than 405 bytes, the peak performance of LD starts to show better performance than the others. The better performance is mainly due to the fact that in LD's peak performance measurement no cleaning was performed.

The peak at 256 KB for all lines is due to the use of direct segments, which enables LD to write data directly to the storage area instead of having to write them into the log area first. Strangely, due to a still unexplained anomaly, the peak performance graph of

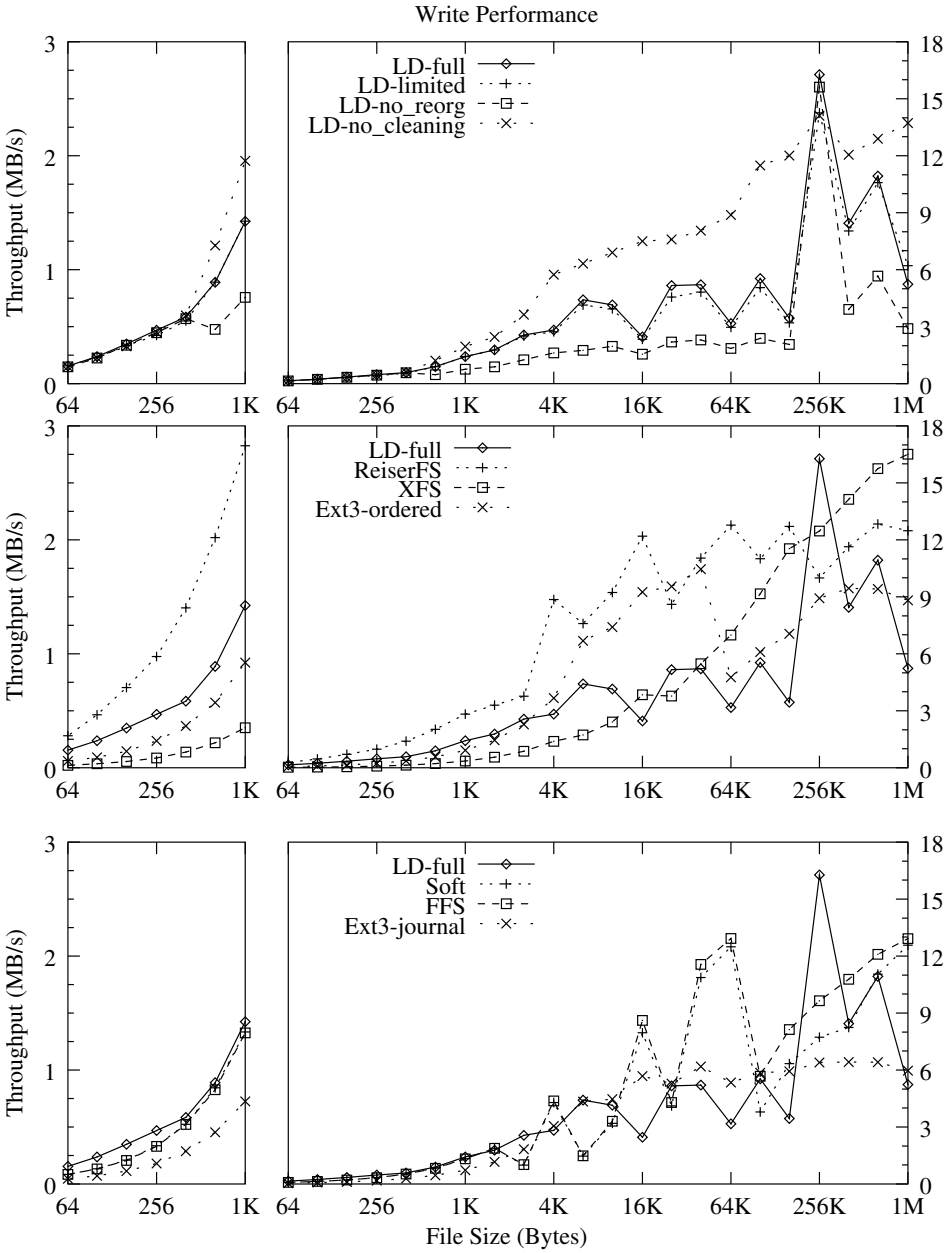


Figure 9.6: Results of the write test-phase. The disk had write-caching enabled and used tagged command queuing with simple tags.

LD shows a lower peak value than the other versions of LD in this graph. For all file sizes smaller than 256 KB, LD writes the data twice; first in the log, and later in the metadata area (immediate files) or in the storage area. For file sizes larger than 256 KB, the parts that do not fit in a multiple of 256 KB segments are written via the log, and therefore, those data are written twice. Furthermore, if data are written in the log, the in-core log segment fills up more quickly, and therefore, when files are not an exact multiple of 256 KB, more log segments are written than if files are an exact multiple of 256 KB. This reason explains the drop in performance after 256 KB. The drop for LD-no_cleaning is smaller than the drop for LD-no_reorg, LD-limited, and LD-full because LD-no_cleaning does not clean the log, and therefore, does not write data twice, but only incurs the cost of extra seeks due to more log segments being written. The performance subsequently rises again for larger file sizes as the tail of the file that is written twice forms an increasingly smaller part of the file.

The graphs for LD-no_reorg, LD-limited, and LD-full also show dips at 16 KB, 64 KB, 160 KB, and 1 MB. These dips are due to these versions of LD needing to resize some areas on disk while these tests were running. In all the other test runs, resizing was not necessary during the test itself. However, given the relatively consistent values in the graphs, we are confident that the performance of all three versions of LD would not have shown those dips, if resizing was not done. Since, in practice, the size of a file system is often relatively stable or changing only slowly, LD can anticipate the need for a resize, and LD can therefore perform the resize in the background. In contrast, in our write test the amount of data in the file system is quickly increased by up to 50%. Consequently, the loss in LD's performance due to resizing will normally not be so noticeable to clients as it is in our performance measurements.

The second pair of graphs show the write performance of the file systems that log only metadata. ReiserFS and LD perform relatively well on small files. They achieve this good performance by storing file data compactly on disk. Both ReiserFS and LD use a tree structure to store data. ReiserFS stores the data of all files (small and large) in a tree and LD stores the data of small files into file-headers which are stored in LD's Mapping, which is implemented as a tree. The other file systems do not store the data of small files so compactly; they write at least one fragment (1 KB) or one block for files that are only tens or hundreds of bytes large. For example, when filling 50,000 files with 64 bytes each, which is only 3 MB worth of data, the file system Ext3-ordered writes over 200 MB. The explanation is that Ext3-ordered uses a block size of 4 KB (50,000 files times 4 KB each is 200 MB). Even though XFS also uses 4 KB blocks, it writes even more than Ext3-ordered: 400 MB. Why it writes this much is unknown to us. The graph of ReiserFS shows peaks at multiples of 4 KB, which can be explained by the fact that ReiserFS uses a block size of 4 KB.

Since Ext3-ordered uses 4 KB blocks and each i-node has room for only twelve direct addresses, an indirect block is needed for files larger than 48 KB. This indirect block explains the sharp drop of Ext3-ordered for 64 KB files. The overhead caused by the seeks to indirect blocks more than halves the throughput of Ext3-ordered.

The bottom pair of graphs show the write performance of FFS with and without soft updates and Ext3-journal. As expected, FFS and Soft perform similarly. Since this write test is dominated by normal data updates, the ability of Soft to write metadata updates

asynchronously has little influence on the write test. The peak at 16 KB for the performance of FFS and Soft are due to the block size of 8 KB that they use. Again, the sharp drop at 101 KB is partly due to the indirect block that is necessary for files larger than 96 KB in FFS and Soft. Another reason for the sharp drop, however, became apparent after performing an additional test run with 96 KB files, the largest size for files that do not need an indirect block. Surprisingly, the performance for writing 96 KB files already showed a significant drop compared to 64 KB files. Analysis of the disk traces revealed that the drop is due to the way FFS and Soft try to write data in clusters of 64 KB. Fitting writes of 96 KB into clusters of 64 KB, somehow, results in write patterns that yield poor performance. This optimization was first introduced in UFS by McVoy and Kleiman [1991] to achieve extent-like performance in file systems that do not support extents. Bad write performance, however, does not automatically result in bad read performance. The write algorithm allocates blocks on disk such that data blocks are clustered on disk, so that the read performance when reading those blocks can be good, as we will see in Section 9.5.5, where we discuss the results of the read test-phase.

The performance of Ext3-journal, the only tested file system other than LD with client data logging, shows a slow rise in performance up to 16 KB. After that, it performs more or less at a same level. For small files up to 4 KB, LD-full clearly outperforms Ext3-journal. For most other file sizes, LD-full and Ext3-journal perform similarly. The dips around 16 KB, 64 KB, 160 KB, and 1 MB for LD-full are due to reorganizers doing a resize during the test (see above). Given the assumption that a resize can be done in the background, if necessary at all, we expect LD-full to perform at the same level as Ext3-journal. The benefit of direct segments can clearly be seen for files of 256 KB and higher.

Analyzing the disk traces showed us that a large part of the time, LD is (randomly) reading the metadata area a single block at a time. This behavior is caused by the *split* and *merge* algorithms of the address-slot table (see Section 8.4.5) in combination with our staccato write method (see Section 6.6). In the write tests, a large amount of data is written that is eventually stored in CDS slots. This process fills up these slots and LD must calculate a new address-slot table to spread data more evenly across the available CDS slots. As explained in Chapter 8, splitting and merging CDS slots is done lazily, which means that the actual moving of data from one slot to another is done by reorganizer processes at a later time. The activity of these reorganizers are not measured in these write tests. Updating the address-slot table, however, is not done lazily, and requires reading parts of the Mapping to discover new split points for the address-slot table. Reading the Mapping is what causes the many reads of the metadata area. Unfortunately, since LD writes metadata using the staccato technique, the metadata are not clustered well, and therefore, reading the metadata lowers the performance of all the three tested versions of LD. Some additional experiments showed that removing these metadata reads could yield a maximum performance improvement of 30%. Therefore, further optimization of the process of updating the address-slot table seems to be worthwhile.

With some reorganization, LD currently reaches a maximum of 4 to 5 MB/s for files smaller than a direct segment (i.e., 256 KB). As argued before, if we can achieve a 30% performance increase by improving LD's metadata performance, LD's write performance would rise to over 6 MB/s. Given that LD writes data twice, the actual bandwidth of the

disk used would be twice this amount, which would therefore be over 12 MB/s. This throughput would be over 64% of the maximum bandwidth of the disk (see table 9.3 on page 236), and would rival the performance of ReiserFS.

In order to compare the write performances of the tested file system more easily, we calculated a weighted average write performance for each file system. This write performance number is a weighted average of the throughputs achieved by a file system for the different file sizes as depicted in Figure 9.6; the weights are determined by the file size distribution found by Gordon Irlam, as summarized in Table 9.6. The resulting weighted averages are shown in Table 9.9, and a graphic representation of the results is shown above the table. These results show that, even though LD-full does not have the best average write throughput, it does have a write performance comparable to FFS, Soft, and Ext3-journal, which gives LD a competitive overall write performance.

In conclusion, it is clear that LD and Ext3-journal lose some performance due to their improved user data integrity guarantees. For small files up to 4 KB, however, LD-full still manages to perform competitively to other file systems; only ReiserFS shows significantly better performance. Furthermore, direct writes help raise LD-full's performance for large files to a very competitive level.

9.5.5 Read Test-Phase

Figure 9.7 shows the results of the read test-phase. The top pair of graphs show that reorganizations in LD are very beneficial for file sizes larger than 10 KB. As can be expected, full reorganization is more effective than limited reorganization. The peak at 256 KB for all three versions of LD shows the effectiveness of direct segments. The throughput reaches almost 17 MB/s, which is 90% of the disk's maximum bandwidth.

The read performance of small files is not much influenced by reorganizations. The reason is that LDFS uses immediate files, which means that small files (i.e., smaller than 444 bytes) are stored in file headers. Therefore, the read performance of these small files is dependent on LD's metadata read performance, which is currently not strongly influenced by reorganizations. Furthermore, files up to and including 4 KB consist of only a few disk blocks. Even though LD without reorganization is not always able to place the blocks of these small files consecutively on disk, it can usually place them close to each other on disk. With the read-ahead automatically performed by the firmware of the disk itself, reading these small files can still be done relatively efficiently without having to seek to every block separately.

The next pair of graphs show the read performance of the metadata logging file systems. ReiserFS's performance for small files is good, which confirms its author's claim that ReiserFS focuses on the handling of small files. For larger files, the spikes show ReiserFS's preference for files that are a multiple of 4 KB. The performance of XFS shows an almost monotonically increasing line. XFS is a slow starter, but ends impressively for large files, reaching 17.3 MB/s, the highest score in the test. The performance of Ext3-ordered is an average performer, reaching almost 12 MB/s for large files. The relatively bad performance of both XFS and Ext3-ordered for very small files is due to the relatively large amount of data that is read, which is probably due to the fact that they use a minimal block size of 4 KB.

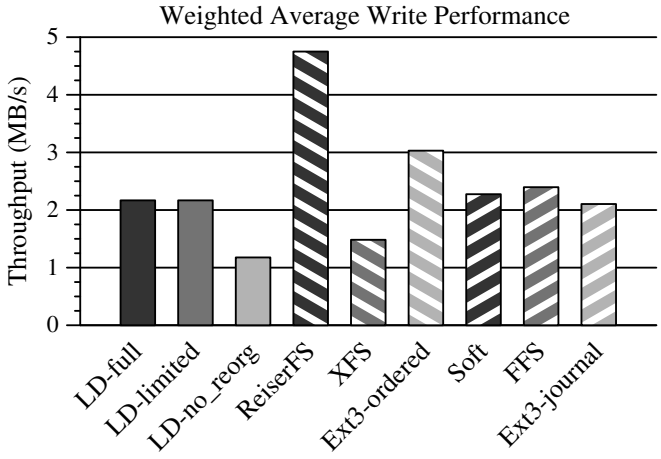


Table 9.9: Weighted Average Write Performance. The disk had write-caching enabled and used tagged command queuing with simple tags.

File system	Write throughput (MB/sec)	% of perf. LD-full
LD-full	2.28	100
LD-limited	2.17	95.1
LD-no_reorg	1.18	51.7
ReiserFS	4.75	208.5
XFS	1.48	65.1
Ext3-ordered	3.03	133.0
Soft	2.27	99.8
FFS	2.40	105.2
Ext3-journal	2.10	92.3

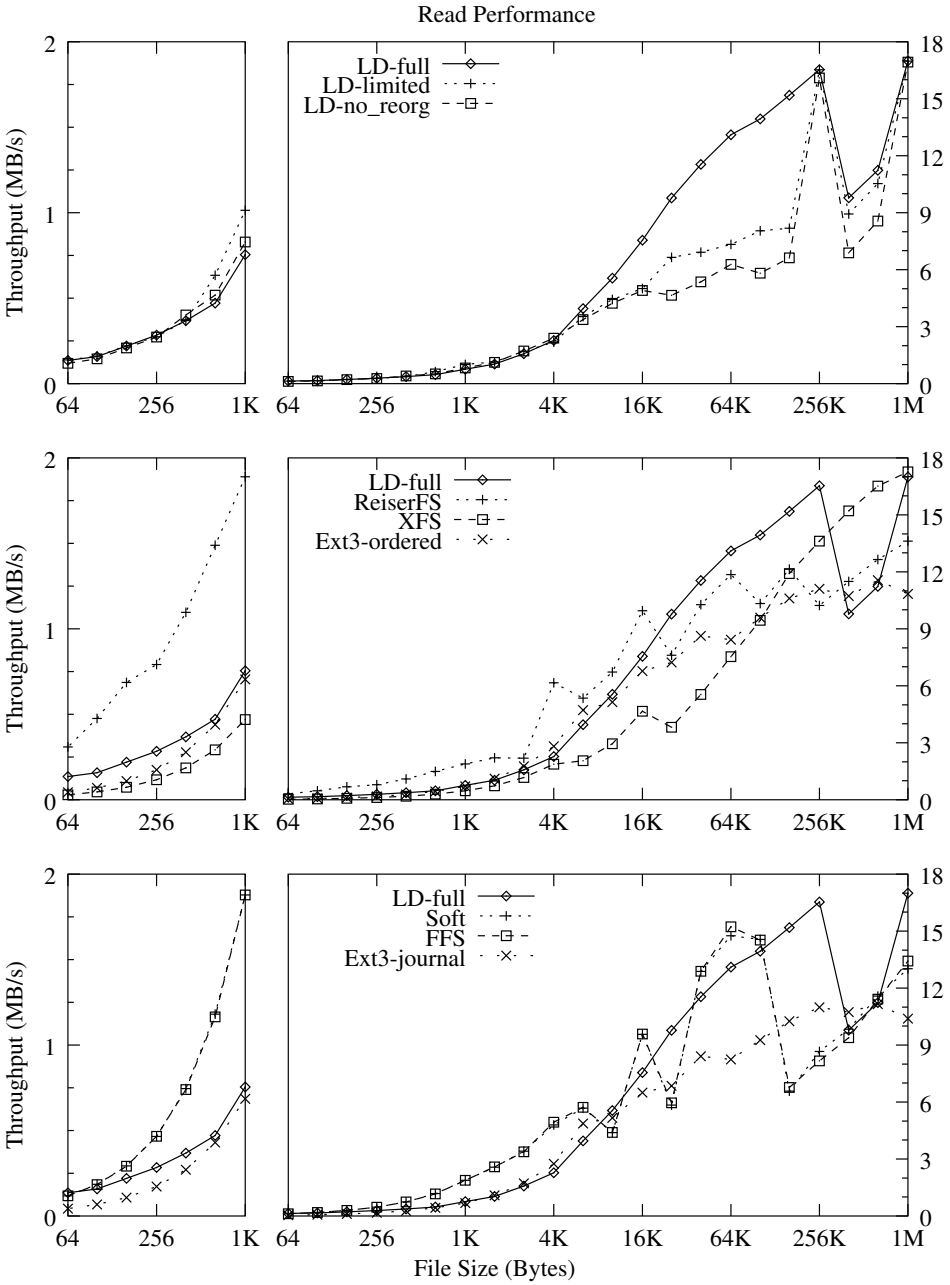


Figure 9.7: Results of the read test-phase. The disk had write-caching enabled and used tagged command queuing with simple tags.

Again, the read performance of FFS and Soft in the bottom pair of graphs are almost identical. The peaks show the preference for 8 KB blocks of both FFS and Soft. However, unlike XFS and both versions of Ext3, FFS and Soft use fragments of 1 KB, which results in good read performance for small files. As expected, there is a drop in the read performance for 101 KB files for FFS and Soft, which is again caused by reading the indirect block. However, the drop in performance is not as large as it was in the write test (see Figure 9.6). The disk traces revealed that, in this case, FFS and Soft have still clustered data blocks and indirect blocks so that in the total test relatively few seeks are required to read them. In contrast, the order in which those blocks were written during the write test required many seeks, which resulted in poor write performance.

The real impact of the seek to read an indirect block can be seen in the read performance of 160 KB files. In this case, almost every file required two extra seeks; one to read the indirect block and one to read the rest of the file's data blocks, whose addresses were stored in that indirect block. The read performance for files larger than 160 KB recovers as the relative cost of the seek gets smaller compared to the total amount of file data read. Ext3-journal, not surprisingly, shows performance similar to Ext3-ordered in the previous pair of graphs.

The relatively good read performance of FFS and Soft is due to their success in clustering files, even on an aged file system. This success, combined with the fact that they keep the metadata close to the actual data in cylinder groups, results in good performance. Even though LD-full clusters the files perfectly on disk, LD-full's performance is lower than FFS's and Soft's performance. This difference is due to the following two reasons.

First, LD's metadata performance slows its overall read performance. All metadata is stored in the metadata area, which on average requires a longer seek to reach than the seek necessary in FFS to read metadata in a cylinder group. Moreover, currently the metadata in LD is not clustered in the metadata area. Consequently, the read-ahead and read cache of the disk cannot make accessing metadata faster. Improving LD's access to its metadata would improve LD's performance substantially. For example, a metadata reorganizer process could be introduced to cluster LD's metadata (see also Section 9.7).

Second, the data files in our tests were spread all over the CDS area, which also increased the seek distance to reach them. The data were so wide spread because the tests created files in different directories (20 files per directory), and moreover, the current implementation of LDFS used an almost random cluster_id for each new directory. Consequently, the contents of each directory were stored in a different CDS slot, if the files were smaller than 256 KB. As a result, when the read test-phase read all files sequentially, LD seeked all across the CDS area. Choosing a more appropriate algorithm to assign cluster_ids to directories would certainly improve LD's performance. However, as the interfaces of the tested file systems do not include support for clustering directories, we did not include this support in LDFS either. With such support, LDFS could have chosen the cluster_ids more carefully to keep the directories more clustered on disk.

FFS and Soft, on the other hand, kept multiple subdirectories and their files in one cylinder group, lowering the seek distances greatly. Recall that our tests created files at the bottom of a three-level directory hierarchy. Each directory contained a maximum of 20 files or subdirectories, except the root directory which could contain up to 125 subdirectories. Analysis of the disk traces showed that FFS and Soft created each subdirectory of the

root directory in a different cylinder group. However, the directory tree created beneath that subdirectory is created in the same cylinder group as that subdirectory. Consequently, as our read test-phase reads the files one directory at a time, in a depth-first-search order, the seeks required to read the files and directories are small for FFS and Soft.

Similar to what we have done with the write performance results, we have also calculated the weighted average read performance for each tested file system. These weighted averages are shown in Table 9.10, and a graphic representation is shown above the table. These averages show that ReiserFS, Soft, and FFS perform similarly, as do Ext3-ordered and Ext3-journal; XFS falls a little behind the others. LD-full has a read performance that is between the performance of ReiserFS and Ext3-journal, giving it a good overall read performance that is competitive to the read performance of other file systems.

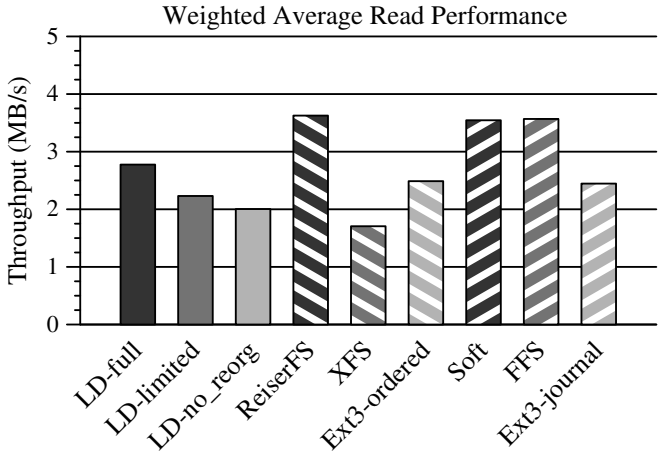


Table 9.10: Weighted Average Read Performance. The disk had write-caching enabled and used tagged command queuing with simple tags.

File system	Read throughput (MB/sec)	% of perf. LD-full
LD-full	2.78	100
LD-limited	2.23	80.4
LD-no_reorg	2.00	72.2
ReiserFS	3.63	130.6
XFS	1.71	61.5
Ext3-ordered	2.49	89.6
Soft	3.55	127.7
FFS	3.57	128.5
Ext3-journal	2.45	88.1

9.5.6 Delete Test-Phase

The last test phase deleted the created files again. The results of this test are shown in Figure 9.8. This test is dominated by metadata updates. In the top graph, we can see that the delete performance of all three versions of LD are almost identical. Unfortunately, after analyzing LD's disk traces, we found that LD's tests for up to and including 4 KB files all suffer from a thrashing metadata cache. Our prototype used a small cache of only 8 MB to cache metadata blocks, which is small compared to the total amount of main memory available in our test machine. Since this delete test is very metadata intensive, the metadata cache was thrashing in the test runs that deleted 50,000 files.

To get an idea of how LD would perform without a thrashing metadata cache, we performed additional experiments for LD without reorganizing in which we increased the metadata cache to 32 MB. The results of these experiments indicate that the delete performance for files over 405 bytes but up to and including 4 KB is approximately 2,000 files/sec, doubling LD's performance. Furthermore, with this larger metadata cache the delete performance for files from 64 bytes to 405 bytes results in a graph decreasing from 1,700 (for 64-byte files) to a little under a 1,000 files/sec (for 405-byte files); another considerable improvement. We expect that LD with limited and full reorganization will show similar performance improvements after increasing the metadata cache size to 32 MB.

The decline in performance for files up to and including 405 bytes can be explained by the fact that the contents of such files are kept in the Mapping in the form of file headers. Therefore, the larger the files, the more disk blocks they occupy in the Mapping, which all have to be accessed by LD in order to delete the files. This effect results in a decreasing performance until the point where the files are so large that their contents are not stored in file headers anymore (i.e., starting from 640 bytes in our tests).

The second graph shows the good delete performance of ReiserFS for files up to and including 16 KB. The performance of XFS is low due to the amount of data it writes, which is probably log data. For example, to delete 50,000 files, XFS writes over 400 MB of data. The amount of data written seems mainly a fixed overhead cost per file, independent of its size. The delete performance of Ext3-ordered suddenly drops after 64 KB due to the indirect block that is being read for files larger than 48 KB.

In the last graph, Ext3-journal shows very similar performance to Ext3-ordered, which was to be expected. The performance of Soft is very stable. Only starting from 101 KB, the performance suddenly drops due to an extra indirect block being read. It is no surprise that the performance of FFS is the lowest of all tested systems due to its (synchronous) metadata updates.

Again, we calculated the weighted average performances for each tested file system. This time, we also included the result of the experiment in which we increased LD's metadata cache from 8 MB to 32 MB. The weighted averages are shown in Table 9.11 and a graphic representation is shown above the table. The table shows that LD's delete performance without the larger metadata cache is not very good, even though it still outperforms XFS and, of course, FFS. However, increasing the metadata cache raises LD's delete performance to a level that is competitive to Ext3-ordered and Ext3-journal.

In conclusion, LD-full's delete performance for files of 101 KB (actually 96 KB) or larger is the best of all tested file systems. Unfortunately, for smaller files, LD does not yet keep up with the delete performance of the other file systems. However, based on some

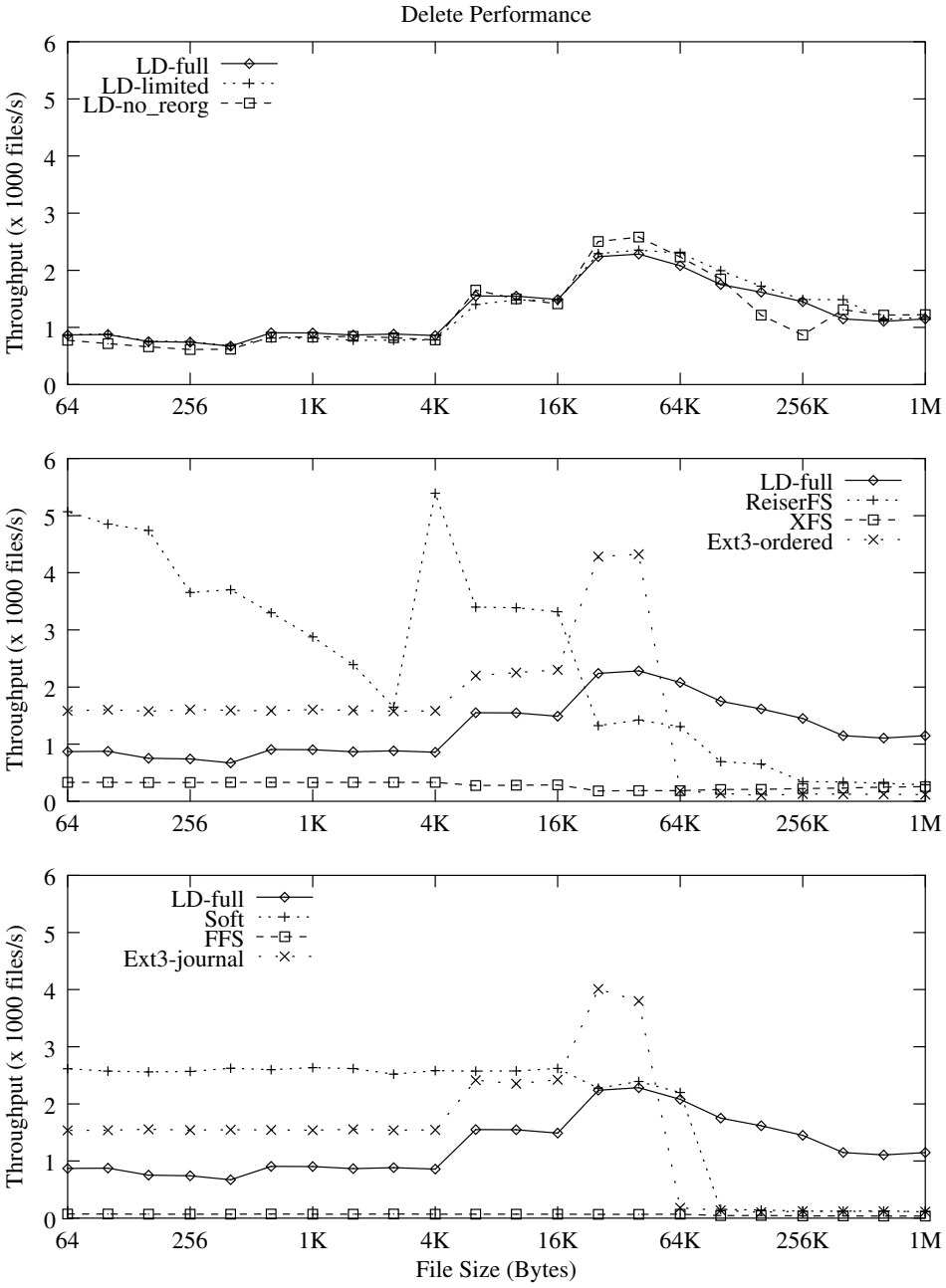


Figure 9.8: Results of the delete test-phase. The disk had write-caching enabled and used tagged command queuing with simple tags.

measurements with a larger metadata cache it seems reasonable to expect that LD's delete performance comes close to being competitive to the delete performance of the other file systems.

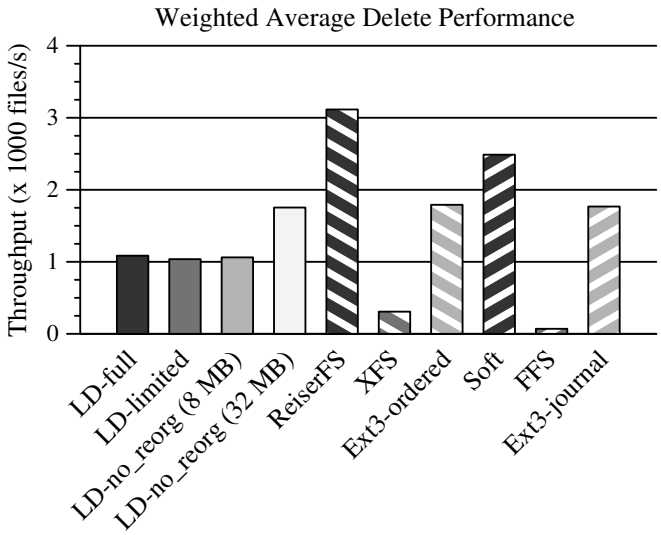


Table 9.11: Weighted Average Delete Performance. The disk had write-caching enabled and used tagged command queuing with simple tags.

File system	Delete throughput (files/sec)	% of perf. LD-full
LD-full	1,085	100
LD-limited	1,037	95.6
LD-no_reorg (8 MB)	1,062	97.8
LD-no_reorg (32 MB)	1,754	161.6
ReiserFS	3,115	287.1
XFS	308	28.4
Ext3-ordered	1,791	165.1
Soft	2,487	229.2
FFS	70	6.4
Ext3-journal	1,768	162.9

9.5.7 Crash Recovery Experiment

The last results we present are the results of the crash recovery experiment as described at the end of Section 9.4.7. Table 9.12 shows the results of this test. Each crash test was repeated three times. Since the amount of work required to perform recovery depends on the state the file system was in when the crash occurred, there is, in general, no single answer to the question how fast a file system can recover. We, therefore, show a time range which show the slowest and fastest times of our three crash tests.

Table 9.12: Timing results of the crash recovery test. The disk had write-caching enabled and used tagged command queuing with simple tags.

File system	Recovery Time (sec)
LD-full	2.6 – 2.9
ReiserFS	2.4 – 2.8
XFS	3.0 – 4.4
Ext3-ordered	0.7 – 0.8
Soft	20.3 – 20.5
FFS	20.5 – 20.6
Ext3-journal	11.4 – 13.4

As can be seen, most systems recover the one GB file system under five seconds. The FFS and Soft file systems run the `fsck` program. In principle, Soft does not need to run `fsck` in the foreground. The soft-updates technique guarantees the integrity of most metadata on disk. The only minor, noncritical inconsistency that may still exist is that some blocks may be incorrectly marked as 'used' in the file system, which results in the file system reporting less free space than it actually has. This inconsistency, however, can be easily corrected by running the `fsck` program in the background. Unfortunately, our current FreeBSD system does not support the background checking, and therefore, we include the time of running `fsck` in the foreground here.

As mentioned above, the time required to recover a file system can vary in practice, depending on a number of factors. For example, the recovery time in LD depends on how many log segments have been written since the last checkpoint. For FFS and Soft, the amount of files and directories on disk influence how long `fsck` takes to check and possibly correct the file system. Our experiment only simulated one type of crash, and therefore, we cannot draw a hard conclusion stating that one file system recovers faster than another in general. However, our experiment does somewhat support our claim that LD can guarantee both client data integrity as well as metadata integrity, while still providing fast recovery.

9.6 Reorganization Overhead

In the performance results discussed above, we have compared the performance of the combination (LD + LDFS) to other file systems. However, the results for LD-full and LD-limited did not include the time that the reorganizer processes needed to cluster the data on disk. In this section, we speculate about whether it is realistic to assume that there is sufficient time in between periods of bursty disk traffic for reorganizer processes to do their work.

One reason for not including the reorganization times is that the LD prototype does not yet include implementations of reorganizer processes that are able to run in the background, let alone efficient ones (see also Section 9.4). Another reason is that, if disk traffic is sufficiently bursty, the reorganizer processes are able to do most of their work in periods of low disk traffic. Therefore, by not including the overhead of the reorganizer processes in the performance results, as we have done in our performance measurements, we get an indication of how the performance of a final implementation of LD-full and LD-limited will compare to the performance of other file systems in practice.

In our experiments, for LD-full and LD-limited a simple reorganizer process was run after aging, after the write test-phase, and after the delete test-phase. The amount of time this simple reorganizer process took to complete its job varied for each test run. The real time necessary for the reorganizations is summarized in Table 9.13.

Table 9.13: The amount of real time required by the reorganizations after aging, after the write test-phase, and after the delete test-phase. The disk had write-caching enabled and used tagged command queuing with simple tags.

Reorganization Times			
File system	**After aging**	**After write test-phase**	**After delete test-phase**
LD-full	24 min.	2 – 30 min.	1/2 – 8 min.
LD-limited	11 min.	2 – 19 min.	1/2 – 6 min.

For LD-full, the real time varied between half a minute and thirty minutes. For LD-limited the real time varied between half a minute and nineteen minutes. Optimization of the reorganizer processes may improve these times. For example, the current simple reorganizer process scans the entire disk to look for work. A more intelligent reorganizer would keep track of where things on disk have changed so that it could focus on the places on disk that need reorganizations.

The numbers in the table suggest that the reorganizer processes in LD-limited are about a third faster than in LD-full. A little surprisingly, reorganizing the entire disk after the aging process did not show the longest reorganization times; they took only 24 and 11 minutes for LD-full and LD-limited, respectively.

If the times in the table are indicative of the times that reorganizations will take under more realistic circumstances, then the overhead of reorganizations will be quite manageable in practice. For example, if we choose to reorganize the disk once per day, the time

overhead would be only 2-3% for LD-full and less than 1% for LD-limited. Therefore, this observation suggests that if the disk is idle for 2-3% of the day, then it would be possible to perform all reorganizing during idle times. In that case, the reorganizers can do their work without interfering with client requests. For instance, most computers will have sufficient idle time at night for the reorganizers to do their work.

Ideally, the reorganizer processes keep the disk reorganized continuously. The advantage of continuously reorganizing the disk is that LD can benefit immediately from the improved clustering. A disadvantage, however, is that reorganizing the disk continuously in idle time is expected to cost more work than reorganizing once a day (for instance, at night). In other words, spreading the work will likely add extra work. Nevertheless, even if, after improving the reorganizer processes, it would still require three times as much work, there still only has to be 10% of idle time spread over the day to keep the disk well organized during the whole day.

9.7 Metadata Performance

During the analysis of LD's performance results, we found that LD's metadata performance was slowing down LD's overall performance. We already mentioned this in the discussions of the write and delete tests (Sections 9.5.4 and 9.5.6, respectively). The problems with LD's metadata performance can be characterized by:

- A metadata cache that is too small, which causes thrashing in our tests.
- No clustering in the metadata area, which causes accesses to logically sequential metadata blocks to require seeks.

The first problem concerns the fact that the current prototype of LD uses a metadata cache that is only 8 MB. This small size resulted in a thrashing metadata cache in some of our performance tests. Since most of the time LD requires a separate seek and a rotational delay to read a metadata block, a thrashing metadata cache has a severe impact on LD's performance.

The second problem is caused by our current design. Recall from Chapter 6 that metadata is written into the metadata area via the *staccato method*. This method enables data to be written to disk fast, but it will not maintain the clustering of the data on disk. Our assumption was that physical clustering is not very important for metadata as it is usually not accessed sequentially in large amounts.

Unfortunately, during the performance measurements, the current implementation of the prototype did perform sequential accesses to its metadata. However, a significant number of these sequential accesses did not originate from the performance experiments themselves, but from LD's own cleaner and reorganizer processes. For example, during the maintenance of the address-slot table, which manages the distribution of client data in the CDS area (see Chapter 8) the Mapping was read sequentially. Consequently, the tests in which a lot of data is written into the CDS area suffered from the overhead of bad sequential accesses to LD's metadata. This effect was visible, for instance, in the write tests.

Fortunately, a number of relatively small changes to LD's design can be made to overcome LD's metadata performance problem. Of course, the design of the cleaner and reorganizer processes can be improved to lower the number of sequential metadata accesses. In addition, other changes can be made to increase LD's metadata read performance. These changes include:

- Increase the size of the metadata cache.
- Increase the size of metadata blocks from 4 KB to 8 KB or even 16 KB.
- Introduce a reorganizer task that clusters metadata within the metadata area in the background.

We have not yet fully evaluated how successful these changes would be. We did, however, perform a few additional experiments in which we increased the metadata cache to 32 MB, which is not overly large compared to the total amount of available memory in the test machine. The results of these additional experiments were reported in Section 9.5.6, and they showed significantly increased performance. Other preliminary experiments with metadata clustering also showed improvements. Based on these results, we are confident that the proposed changes will improve LD's performance considerably and thus make LD's performance even more competitive to the performance of other file systems than it already is.

9.8 Summary

In this chapter, we evaluated the design of LD by examining how well LD achieved its three goals of improving modularity, improving data integrity, and still providing performance competitive to other systems. We primarily looked at LD's performance with numerous experiments. However, we also made some observations concerning LD's modularity and data integrity.

The prototype implementation of LDFS shows that a file system on top of LD can indeed be implemented relatively easily. The ARU mechanism provided by LD enables simple programming constructions and provides data integrity guarantees that could otherwise not so easily be provided. The results of the crash tests suggest that recovery can be done within seconds.

The many performance experiments show LD's strengths, but also some weaknesses. Some observations that can be made are:

- Even though LD is only at the beginning of its life-cycle, the performance results suggest that, with some minor enhancements such as a larger metadata cache, LD already provides performance that can be considered competitive to the performance of other file systems.
- Direct segments provide very good write and read performance, which can be seen in the performance numbers for files of 256 KB or larger in the write and read test-phases in Sections 9.5.4 and 9.5.5, respectively.

- Using file headers to implement immediate files provides competitive write performance for small files. This result can be seen in the performance numbers for files up to and including 443 bytes in the write test-phase in Section 9.5.4.

- Keeping data clustered on disk provides excellent read performance, which can best be seen in the read performance measurement in which the entire aged file system was read (see Section 9.5.2). Consequently, an approach that continuously keeps the disk well-organized seems to be worthwhile and deserves further research.

- LD can provide competitive performance even though it uses very small blocks (i.e., 512 bytes) whereas other file systems use considerably larger blocks (4 KB or 8 KB, in some cases in combination with 1 KB fragments). In general, using small blocks requires file systems to keep more administration, and results in more disk accesses due to fragmentation. However, the use of compression in LD's Mapping, and clustering of data blocks can avoid these disadvantages of using small blocks.

- Improving LD's metadata performance will yield significantly better performance. Possible solutions to improve the metadata performance were presented in Section 9.7 and include improving LD's cleaner and reorganizer algorithms, increasing LD's metadata cache, using larger metadata blocks, and introducing a reorganizer task that clusters metadata in the metadata area.

- Putting the tail of large files (i.e., files larger than 256 KB) into the CDS area while the rest is in the CDL area results in a large seek that lowers performance. The larger a file is, the less noticeable the overhead of this seek becomes. This result can be seen in the performance numbers of the write test-phase in Section 9.5.4.

- The idea of running the reorganizer processes during idle time of the disk seems viable. Section 9.6 speculates that 10% of disk idle time spread over the day would be sufficient to keep the disk well-organized continuously.

Based on the above observations, we conclude that it is possible to provide improved modularity and improved data integrity guarantees, while still providing performance competitive to other existing file systems. However, there are some issues that need further research. For example, the reorganizer processes need further research, and real-time experiments with a version of LD that runs within a kernel are necessary to evaluate the overall performance of LD including the overhead of background reorganizers in a more realistic environment. We will come back to these issues in Chapter 11 where we discuss possible future work.

Chapter 10

Related Work

Storage management has been the subject of much research in the past. In this chapter, we look at some of the storage systems and storage techniques proposed in that research. Since much work has been done in the field of storage, it is impossible to discuss all previous work related to storage. The work discussed in this chapter, therefore, discusses only the more common techniques and the systems directly related to LD.

The research into storage management also includes the category of multidisk systems, such as RAID [Chen et al., 1994], and storage systems that are meant to be deployed in distributed computer environments. However, even though the research in this category has produced techniques concerning data integrity and performance improvements, we have left this category out of our discussions. Their basic architecture is too different given that LD currently focuses on a single-host, single-disk system.

The discussion of related work is split into six sections. The first three sections discuss related work with respect to the three problem areas identified in Chapter 2: modularity (Section 10.1), data integrity (Section 10.2) and performance (Section 10.3). Each section discusses common techniques as used in other systems to solve the particular problem area. The last three sections discuss three example systems in more detail: Log-Structured File System, Loge/Mime, and Disk Caching Disk.

10.1 Modularity

In Chapter 2, we argued for the separation of file management and disk management in order to increase the modularity of file systems. Obviously, this is not the only way to add more modularity to a system. In this section, we look at a few other systems that modularize the functionality of storing data on disk in different ways.

LD adds modularity to a system by putting desirable functionality, such as data integrity, into a separate storage management layer. Basically, LD provides a disk block interface, but adds some support for data integrity (i.e., ARUs and streams) and physical clustering (i.e., disk files and disk clusters). A similar storage management layer is offered by systems like Loge [English and Stepanov, 1992] and Mime [Chao et al., 1992]. They also provide a block-level interface and some data integrity features, such as atomic

(multi)block writes. However, they do not support grouping blocks in larger logical units. Consequently, whereas clients can indicate to LD how they want their data blocks clustered on disk, they cannot do so with Loge and Mime. We will look at Loge and Mime in more detail in Section 10.5.

Another method of modularization of the file system is the stackable file system design [Heidemann and Popek, 1994, 1995; Khalidi and Nelson, 1993]. In this modular design, complex filing services are constructed by combining different 'building-block layers'. Each layer adds new functionality, such as compression, encryption, or remote access. The authors argue that stacking enables new file system technology to be more easily integrated with existing techniques, which aids the broad acceptance of the new technology. The principle of layering was already present in the V-Node architecture [Kleiman, 1986].

The main difference between LD's modularization and the stackable file system layer design is that the latter provides a more general framework to add functionality to systems in a modular design. In particular, it does not dictate how the functionality should be split up, but simply offers a framework that describes how the layers can cooperate to form one complex file system.

The exokernel operating system architecture [Engler et al., 1995; Kaashoek et al., 1997], provides applications direct and secure access to hardware resources, which is claimed to increase performance. In this architecture, a minimal kernel (*exokernel*) securely multiplexes available hardware resources by separating protection from management: resources are protected by the kernel, but management is delegated to applications. For example, in one implementation of the exokernel, called *XoK* [Kaashoek et al., 1997], the disk subsystem, called *XN*, is responsible for safely multiplexing disks among multiple file systems running on top of XoK.

XN does not do the data storage itself, but only protects disk blocks from unauthorized access and guarantees that disk updates are ordered according to the rules of Ganger and Patt [1994] to keep file system integrity across crashes (see Section 10.2.3). Other abilities like allocation policies, disk block layout, and recovery semantics are left to the file system. The exokernel architecture, similar to stackable file systems, also provides a more general solution than LD's modularization. Other functionality, such as LD's data integrity and performance improvements are left to a file system on top of XoK.

10.2 Data Integrity

In the past, many solutions have been proposed to solve the problem of maintaining data integrity in the face of crashes. Some of these solutions are used by file systems to ensure that file system integrity is maintained. However, most file systems consider the consistency of the file system *structure*, as represented by certain metadata such as i-nodes and directories, more important than the consistency of the actual *data* of users stored in the file systems. These systems ensure that only these metadata of the file systems are protected against crashes. Only a few file systems also guarantee the integrity of user data. The reason for most file systems to concentrate on *metadata integrity* only is performance overhead. Below we will briefly look at some of the techniques that file systems use to

maintain the integrity of user data and/or metadata.

10.2.1 Transactions

Transactions have long been the basic unit of data integrity in database systems. The **ACID**-properties of transactions (see Section 2.3.4) offer their users strong data integrity guarantees. Unfortunately, transactions are not frequently used in file systems because of the significant performance overhead of transaction support. Nevertheless, some operating systems do incorporate support for transactions. An example is QuickSilver [Schmuck and Wyllie, 1991; Haskin et al., 1988]. Furthermore, Seltzer [1993] has looked at embedding a transaction manager within a log-structured file system.

LD, as most other mentioned systems, currently does not support full transactions. However, it does support ARUs, which are similar to transactions with limited durability and without full isolation (i.e., serializability). ARUs can be used as building blocks to support full-fledged transactions with respect to disk reads and writes. In the future, LD may extend ARUs to include full isolation (i.e., concurrency control) and durability (see Chapter 11).

10.2.2 Journaling/Write-Ahead Logging

A technique generally used in database systems to guarantee data integrity (e.g., in the form of transactions) is logging. It has, however, also been applied in the field of file systems. For instance, logging is used in Ext3 [Tweedie, 2000], Episode [Chutani et al., 1992], Cedar [Gifford et al., 1988; Hagmann, 1987], ReiserFS [Reiser], XFS [Sweeney et al., 1996], and JFS [JFS; Best, 2000]. The Log-Structured File System (LFS) uses logging in an extreme form: the whole disk is a log. We will discuss LFS in more detail in Section 10.4. In the course of time, different names have been used to refer to this technique, such as journaling, write-ahead logging, or simply logging. We will use these three names interchangeably.

The general technique of logging has been explained in previous chapters. In short, each operation generates a log entry in a log on disk. After a crash, this log is used to undo and/or redo operations in order to bring the system into a consistent state. In contrast to LD, however, most of the above mentioned file systems, by default, only support metadata logging: Episode, Cedar, ReiserFS, XFS, and JFS. Note that Cedar does not need user data logging since it only supports immutable files. With metadata logging only, the data integrity of user data is not guaranteed. As a result, a crash could, for example, cause incorrect data (e.g., data belonging to already deleted files) to appear in files.

Ext3, however, does support user data logging. It is the standard logging file system of the Linux operating system. Ext3 offers three levels of logging. The desired level of logging (*journal*, *ordered*, or *writeback*) can be selected at mount time. The highest logging level 'journal' does both user data and metadata logging. 'Ordered' does metadata logging, but also groups changes to user data and the corresponding metadata together. When writing data to disk, user data is written to disk before the corresponding metadata. The result of this ordering is that metadata always point to valid user data. The lowest logging level 'writeback' does metadata logging only, and is comparable to the type of

logging used in ReiserFS, XFS, and JFS. An additional advantage of Ext3 is that it is backward compatible with Ext2 [Ext2FS; Card et al., 1994; Beck et al., 1998], which facilitates its acceptance.

The logging technique in LD also differs on another point in the way logging is used in most file systems mentioned in this subsection. With the exception of LFS, the other systems in this subsection use the log only during recovery. In LD (and LFS), the log is an integrated part of the overall storage structure. Data written into the log may be used (i.e., read) during normal operation.

10.2.3 Soft Updates

The term 'soft updates' [McKusick and Ganger, 1999; Ganger et al., 2000] refers to a technique that solves the metadata inconsistency problem without using a log. Soft updates enable updates to metadata to be written asynchronously to disk almost (see last paragraph of this section) without endangering the metadata consistency during a crash. Currently, the soft-updates technique is used in several versions of BSD, including the FreeBSD version that we used in our experiments (see Chapter 9).

With soft updates, the file system keeps track of dependencies between updates to metadata blocks. These dependencies indicate the order in which the metadata blocks must be propagated to disk in order to safeguard the integrity of the file system structure on disk. For example, creating a new file adds a dependency between the i-node of the new file and the directory entry for the new file, indicating that the i-node must be written to disk before the directory entry. When the file system wants to write a specific metadata block from its main-memory buffer to disk, it first checks to see if there are any unsatisfied dependencies for that metadata block. If so, these dependencies are first removed by temporarily rolling back the updates that were responsible for adding these unsatisfied dependencies, before the metadata block is written to disk. In other words, the contents of the metadata block are temporarily brought back to the state before the updates were done.

The rollback is only performed for the sake of writing a version of the metadata block to disk that is consistent with the other metadata blocks already on disk. Consequently, a crash will always leave a consistent state of metadata on disk. While the metadata block is being written to disk, access to the metadata block is denied so that users do not see the temporarily rolled-back situation. After the block has been written to disk, the rollback is undone and normal access is granted to the metadata block again.

Ganger and Patt [1994] define three ordering requirements which govern when updates must be temporarily rolled back before writing a metadata block to disk:

(1) Never point to a structure before it is initialized (e.g., an i-node must be initialized before a directory references it).

(2) Never reuse a resource before nullifying all previous pointers to it (e.g., an i-node's pointer to a data block must be nullified before the disk block may be reallocated for a new i-node).

(3) Never reset the last pointer to a live resource before a new pointer has been set (e.g.,

when renaming a file, the old name for the i-node must not be removed until after the new name has been written).

The rollback enables the file system to keep the metadata blocks on disk in a consistent state at all times. The only minor inconsistencies that may occur after a crash are blocks or i-nodes that are marked as allocated in the bitmaps keeping track of allocated blocks and i-nodes, but that are in fact free. However, these inconsistencies are not severe and the file system can be used as normal, although it may seem to have less free space available to it than in reality. To fix these last inconsistencies, a program such as `fsck` could be run in the background. For a performance comparison between journaling and soft updates the reader is referred to [Seltzer et al., 2000].

The soft-updates technique focuses only on the consistency of metadata. It does not protect user data, whereas LD uses logging to guarantee the consistency of both user data and metadata. Furthermore, the soft-updates technique, unfortunately, still updates metadata blocks by overwriting them (i.e., in-place updates). Therefore, an unfortunate power failure can still cause serious inconsistencies in a file system protected with soft updates. In contrast, LD avoids such in-place updates, and therefore, does not suffer from said dangers of soft updates.

10.2.4 Versioning File Systems

Versioning file systems keep a version history of their files and directories. Such systems view information as valuable and storage as cheap. Therefore, storage can be used to protect valuable data by keeping older versions. In fact, versioning file systems can offer data integrity guarantees on another level than journaling. For example, versioning file systems can recover from accidental deletions or overwrites by users, which is something not offered by journaling file systems.

However, simply keeping version history is not enough to be able to guarantee data integrity after a crash. The file system still needs a method to update its (meta)data on disk in a safe manner. Versioning, therefore, should be considered an addition to journaling or soft updates. The combination is capable of allowing the users to restore any state of a file, if that state has been preserved as a version.

The Elephant system [Santry et al., 1999], for example, enables its users to choose the desired level of versioning for each file individually. The level of versioning varies from no versioning to retaining every version, and some levels in between. When choosing a level in between, the Elephant system uses heuristics to determine which versions are worthwhile to keep and for how long to keep such versions.

An important problem of versioning file systems is how to keep the size of the metadata manageable. If naively stored, the metadata can quickly become significantly large compared to the actual versioned data. Soules et al. [2003] present a method to efficiently store the metadata to keep track of different versions. The authors describe two methods of storing metadata more compactly: *journal-based metadata*, which stores only the differences between two successive versions of a file's metadata in journal entries, and the *multiversion b-tree*, which retains all versions of a metadata structure within a single tree. The former method is used to store i-nodes and indirect blocks, the latter is used to store directories efficiently.

10.2.5 Persistent Main-Memory Regions

In the previous subsections, we have mentioned techniques that have been integrated with file systems in order to guarantee data integrity. In this subsection, we describe a couple of techniques that provide support for any application to guarantee integrity of their stored data. The systems mentioned below all add certain data integrity guarantees to a region of main memory. This protected region then enables an application to protect any changes to its data, so that it can atomically change from one consistent state to the next.

The lightweight Recoverable Virtual Memory (RVM) system [Satyanarayanan et al., 1994], for example, allows an application to assign regions of its virtual address space to have transactional guarantees. With RVM, applications can atomically update their persistent data structures, such as the metadata of storage repositories. Portability was one of the key issues in RVM's design, and therefore, it has been implemented as a library. Internally, RVM uses a no-undo/redo value logging scheme to implement the transactional guarantees.

Logged Virtual Memory (LVM) [Cheriton and Duda, 1995] offers functionality similar to RVM. LVM is a virtual memory system extension that provides logs of write activity to specified virtual memory regions. Note that, unlike RVM, LVM is part of the operating system's virtual memory system. Except for calls to indicate which virtual memory regions to log, LVM does not require programmers to insert calls into their programs to indicate which memory writes to log; LVM logs every write to the specified virtual memory region.

The Rio File Cache [Ng and Chen, 2001; Chen et al., 1996] focuses on protecting main memory against system crashes due to software errors. The goal of the Rio File Cache is to make main memory as safe as a 'write-through' cache, but without doing the actual writes to make the data immediately safe on disk. The solution used is to provide a special 'safe-sync' reset button that a user can press after a crash, which will dump the main memory to disk. The difficulty is in designing and integrating the safe-sync procedure into an existing kernel, such that it still works after a crash has occurred. The authors have implemented the safe-sync procedure in a BSD kernel and tested their solution by using fault-injection on a running kernel.

The RAPID-cache [Hu et al., 2002] provides a reliable and inexpensive write cache for high-performance disk systems. The designers of the RAPID-cache argue that the use of a single NVRAM cache in high-performance disk systems forms a single point of failure. Using two copies of NVRAM is too expensive. The solution chosen in the RAPID cache is to use a disk as a backup cache, because disks have a higher MTTF (up to one million hours) compared to NVRAM (15,000 hours).

The RAPID cache consists of two redundant write buffers on top of a normal disk system: a primary cache and a backup cache. The primary cache is made up of an amount of normal RAM or NVRAM. The backup cache consists of a much smaller amount of NVRAM on top of a log disk. The backup cache is used to make the primary cache 'safe' efficiently. New data written into the primary cache are immediately written to the backup cache, which temporarily stores the data in its NVRAM. When enough data has accumulated in the NVRAM of the backup cache, it is written to the log disk with a large, and thus efficient, write.

A similarity between RVM, LVM, the RAPID cache, and LD is that they all use a log

to help make updates safe. However, the main difference between them is that LD focuses on disk I/O, whereas the others focus on main memory. Consequently, LD also supports clustering data on disk in order to increase read performance.

10.3 Performance

The last problem area we identified in Chapter 2 was performance. In LD, we use the techniques of collective writes, clustering, and ARUs to improve performance. The ARUs make it possible to avoid the use of slow synchronous metadata updates, which more traditional systems use to guarantee metadata integrity. As we discussed in Sections 10.2.2 and 10.2.3, logging and soft updates are alternative techniques that can be used to avoid synchronous metadata updates, and therefore, improve performance while guaranteeing metadata integrity. In this section, we look at other techniques that are also aimed at improving either read or write performance or both.

10.3.1 Data Block Placement

The amount of disk-arm movements and the amount of rotational delays determine how fast blocks can be read back from and written to disk. Therefore, most file systems try to place the blocks on disk such that the overhead of disk-arm movements and rotational delays is small. Preferably, data that are frequently accessed together should be stored contiguously on disk (i.e., be clustered).

For example, FFS [McKusick et al., 1984] divides the disk into *cylinder groups* and tries to allocate blocks of files in the same directory into the same cylinder group, which minimizes the length of disk seeks when accessing those files. Some file systems allocate files in extents, which are contiguous ranges of blocks, instead of in single disk blocks with the goal to increase performance by grouping blocks into larger physical units. Even though SunOS UFS does not use extents, McVoy and Kleiman [1991] have modified it to group I/O operations in clusters instead of dealing with individual blocks. The result is that it can approximate the behavior of extent-based file systems.

Schindler et al. [2002] utilize disk-specific knowledge to try to place data in track-aligned extents, called *traxtents*. By allocating on disk track boundaries, they can avoid rotational delays and track crossing overhead. The rotational delay can be avoided if data is track aligned and the disk uses *zero-latency access*, which means that the disk can read the sectors of a track 'out-of-order' by using the disk's internal track cache.

The Co-locating Fast File System (CFFS) [Ganger and Kaashoek, 1997] uses the techniques of *embedded i-nodes* and *explicit grouping*. The first technique stores the i-nodes of files in the directory structure itself. Consequently, accessing the data of a file eliminates one level of indirection, as reading the metadata of the file does not require a separate read anymore. The second technique refers to the fact that CFFS allocates the blocks of multiple small files adjacently on disk and accesses them as a unit in most cases. The idea behind explicit grouping is that accessing multiple consecutive blocks is not much more costly than reading a single block, since the overhead of the seek and rotational delay dominates the total transfer time. The advantage of explicit grouping over the cylinder

groups of FFS is that it reduces both seek time and rotational delay since data are accessed as a unit, whereas storing data in cylinder groups only reduces seek time.

Another technique of placing data blocks on disk is called Adaptive Block Rearrangement [Akyürek and Salem, 1995], which reduces disk seek times by copying frequently referenced blocks from their original locations to a reserved space in the middle of the disk. The stream of incoming requests is constantly monitored to estimate which blocks are most likely to benefit from being copied to the middle of the disk. The authors suggest it is sufficient to determine which blocks to copy once a day. Rearranging only 3% of the data can result in 30% to 80% reduction in seek time, depending on the workload.

The above mentioned techniques all place the data on disk without user control. The techniques use heuristics to determine which blocks to place where. On top of FFS-like file systems, however, PLACE [Nugent et al., 2003], which is a user-level library, gives users more control over the file layout. Via PLACE users can place files and directories into specific and localized portions of the disk. It is implemented using gray-box techniques, that is, some general knowledge of how the underlying system behaves or is implemented is combined with run-time observations of the system in order to provide more powerful services. In this case, PLACE uses the knowledge of how FFS allocates blocks into cylinder groups to enable users of PLACE to colocate files and directories that exhibit temporal locality of access.

LD also offers its clients the ability to indicate which blocks should be stored clustered. This approach differs from most techniques mentioned above. Internally, LD could benefit from techniques such as track-aligned extents, in order to make accessing clustered data even more efficient. Furthermore, the technique of explicit grouping could also be employed efficiently within LD. Even though, LD already clusters data of small files together in CDS-slots (see Chapter 8), it does not access the blocks within a CDS-slot in larger units.

10.3.2 Disk Scheduling

The purpose of disk scheduling [Geist and Daniel, 1987; Seltzer et al., 1990; Teorey et al., 1972; Teorey, 1972; Worthington et al., 1994; Worthington, 1995] is to increase disk throughput by reordering pending disk requests in order to lower the seek distances and rotational delays. Several algorithms have been developed, such as Shortest-Seek-First (SSF), elevator, or Shortest-Time-First (STF). With SSF, the request for which the disk head has to travel the smallest distance is served first by the disk. This algorithm results in a lower average response time than if the disk used First-Come-First-Serve (FCFS). Unfortunately, SSF is not a fair algorithm because it may suffer from starvation, which can be seen in the large variance in the response times. The elevator algorithm tries to solve this fairness problem by moving the disk head in one direction only; the result is a smaller variance in response times.

With STF, the next scheduled request chosen is based on both the seek distance and the rotational delay, which requires more accurate information about the location of the disk head. STF achieves a higher throughput than SSF, but suffers from maximum response times that are much worse than with FCFS. Therefore, variations on STF have been devised that weigh in the age of a request when determining which request to schedule next,

in order to achieve fair scheduling.

Iyer and Druschel [2001] discovered a method to improve the effectiveness of disk scheduling algorithms. Disk schedulers, generally, schedule the next request as soon as the previous request has finished. Such a scheduler is called *work-conserving*. Unfortunately, many applications send requests in a synchronous manner, that is, they send the next request only after the previous has been completed. Consequently, the scheduler often incorrectly assumes that the process issuing the last request has no further requests, which is called *deceptive idleness*, and is forced to switch to a request from another process. Applications often send requests that concern data that are near each other on disk, and therefore, the choice to switch to a request of another process may be inferior to serving the next request of the last process. The solution the authors propose is called *anticipatory scheduling* and involves waiting a small period before scheduling the next request which allows the last process to send a next request.

Lumb et al. [2000] have devised *freeblock scheduling*, which utilizes the rotational latency periods for useful media transfers. The authors differentiate between two types of disk requests: low-priority workload and normal workload, called background and foreground, respectively. Freeblock scheduling consists of interleaving background requests in the stream of foreground requests, without effecting the foreground response times. In short, the technique works as follows. For each foreground request that is scheduled to be serviced next, the expected rotational delay is determined. Then, the queue of pending background requests is searched to see if the disk can service a background request ahead of this foreground request, and still be on time to service the foreground request without losing a rotation.

The authors claim that 20%-50% of the bandwidth of a disk can be provided to background applications in this way. Candidates for background work are scanning applications (e.g., data mining), storage optimization (e.g., clustering, cleaning), and prefetching (see also Section 10.3.3).

For freeblock scheduling to work, accurate timing and positioning information must be available, which is usually only available in disk firmware. However, in a follow-up study, Lumb et al. [2002] have designed a way to use freeblock scheduling outside disk firmware.

Although disk scheduling violates one of our data integrity requirements (see Chapter 2) as it involves reordering disk requests over which the user has no control, LD can still safely use disk scheduling most of the time. First, since LD does not guarantee ordering between commands in different streams, LD can use a disk scheduling algorithm to decide which stream to serve first. Second, LD can use disk scheduling to increase throughput during cleaning and reorganizing, which move data from the log to the storage area and cluster data within the storage area, respectively. During recovery, the correct execution order of the recovered commands is guaranteed by the log tuples in LD's log (see Chapters 5 and 7).

10.3.3 Prefetching

Prefetching refers to the technique that tries to increase read performance by anticipating which blocks will be needed in the near future and reading them from disk ahead of time.

Why prefetching works has been researched by Shriver et al. [1999]. For example, when the first block of a file is read, most file systems anticipate that the file is going to be read sequentially and prefetch more blocks from the beginning of that file. The difficulty in prefetching is deciding when and what to prefetch.

Patterson et al. [1995] state that an application's read-access patterns are largely predictable. Therefore, they propose that applications should inform the file system of future demands by providing hints, so that the file system can prefetch and cache data effectively. The authors call this *informed prefetching and caching*.

Griffioen and Appleton [1994] describe *automatic prefetching* which automatically predicts future file system requests based on past file accesses. That previous accesses can accurately predict upcoming file accesses was also shown in Kroeger and Long [2001]. Automatically generating predictions has the advantage that a rewrite of applications or programmer interventions are not necessary. Furthermore, automatic prefetching works across applications. If two applications are often executed one after the other, automatic prefetching will prefetch the blocks of the second application when the first application is run.

Even though automatic prefetching based on static analysis or historical access patterns is successful, Chang and Gibson [1999] claim its effectiveness can be improved for applications with irregular and input-dependent access patterns. Therefore, the authors propose to automatically generate I/O hints through speculatively pre-executing the application's code. In short, each program is transformed so that it runs two versions (threads) of the same program. One thread is the main thread, the other is a speculative thread, which only runs when the main thread blocks on I/O.

The purpose of the speculative thread is to find future read accesses and to generate hints to the underlying file system. It does not do any actual I/O. For this scheme to work, the speculative thread must be synchronized with the main thread. If the speculative thread is 'off-track', it would generate incorrect hints. Whether the speculative thread is off-track can be determined by comparing the hints generated by the speculative thread with the actual I/O's generated by the main thread. If they do not match, the speculative thread is off-track and is restarted with the current state of the main thread.

Prefetching has not been implemented in LD yet. LD has read-ahead calls in its interface, but their usefulness has not yet been examined. The read-ahead calls are, in fact, user-initiated hints to LD that it should prefetch certain data. The techniques of automatically determining data to prefetch could also be used within LD, in addition to the read-ahead calls already present.

10.3.4 Caching

Caching has long been used to improve the performance of storage systems. Its purpose is to keep previously referenced data in main memory for future reference. In combination with prefetching, it must also hold prefetched data for future use. There are two ways to increase the effectiveness of the cache. First, the size of the cache can be increased so that it can hold more data. Unfortunately, research has shown that increasing the amount of cache memory quickly shows diminishing returns [Baker et al., 1992]. Some systems, therefore, focus on the other method to increase the effectiveness of the cache, which is

choosing the correct cache replacement policy. A cache replacement policy is used to decide when to evict data from the cache to hold other data. The most common policy is the Least-Recently-Used (LRU) policy, which evicts blocks on the basis of how long it has been since they were last accessed.

However, the LRU policy is not always the best choice. Cao et al. [1994, 1996] propose a scheme where file caching is under control of applications. Instead of the kernel choosing a cached block to be evicted, the kernel chooses a candidate block, but the application owning the block can overrule that decision and choose another one of its blocks instead.

The DEAR scheme [Choi et al., 2002] automatically detects block reference patterns of applications and dynamically decides whether to use a Least-Recently-Used (LRU), Most-Recently-Used (MRU) or Least-Frequently-Used (LFU) policy.

RAPID (which has been discussed in Section 10.2) and Disk Caching Disk (DCD) [Hu and Yang, 1996; Nightingale et al., 1999] try to make caching safe and efficient by writing the contents of the cache to disk using logging techniques and collective writes. We will discuss the Disk Caching Disk in more detail in Section 10.6.

LD's prototype also has a cache to hold recently accessed data blocks, which is not only beneficial for clients but also for LD itself. For example, LD's cleaner processes can benefit when segments that must be cleaned are still cached in main memory. Cleaning segments still present in the cache is obviously more efficient than cleaning segments that must first be read from disk. Therefore, in LD, a cache replacement policy that favors keeping uncleaned segments cached may be beneficial. In the future, techniques to change the used cache replacement policy, either under the control of applications or automatically as mentioned before, could also be applied within LD.

10.4 Log-Structured File System

In this and the following two sections, we discuss three specific systems in more detail. We start with the log-structured file system (LFS). In 1989, Ousterhout and Douglis [1989] proposed a new scheme of storing data on disk, in an attempt to beat the I/O bottleneck. They predicted that disk traffic would become write dominated because reads would be satisfied by large caches. Therefore, the solution they presented was a write-optimized file system, called the log-structured file system. Unfortunately, their prediction that reads would be satisfied from the cache has not come true. Although cache sizes have increased, the capacity of disks and the amount of data stored on them have increased more, and consequently, good read performance is still required of a file system as not all reads can be satisfied from the cache.

In LFS all modifications are written sequentially to disk in a log-like structure, which makes random writes very efficient. To increase the utilized bandwidth of the disk, LFS writes data to disk in large segments. In addition, LFS makes crash recovery faster since all the latest changes to the disk are located at the head of the log. Unfortunately, the log-like structure of LFS degrades sequential read performance in LFS.

Similar to other more traditional file systems, such as FFS, LFS supports files and directories. Each file and directory has an associated i-node, which keeps track of the

data blocks assigned to them. However, unlike FFS, i-nodes do not have fixed locations on disk, and therefore, an i-node map is used to keep track of the locations of the i-nodes themselves. The i-node map itself is written in a fixed checkpoint region on disk.

In time, the free space on the disk will become fragmented and LFS needs a way to create large enough consecutive free spaces on disk to write new log segments. In LFS, this process is called *cleaning*. Basically, cleaning consists of reading in a number of log segments, identifying the live blocks in those segments and writing them back to disk in a smaller number of new log segments.

Rosenblum and Ousterhout [1992] designed a cleaning heuristic that decides which segments to clean based on the utilization and the age of a segment. Segments with a low utilization are cleaned because that would free up space with the least amount of cleaning. Furthermore, by including age as a factor in deciding which segments to clean, even a small amount of free space in segments with a relatively high utilization can eventually be reclaimed. The goal of this cleaning heuristic is to reach a bimodal distribution of segments on disk in which a segment either has a high utilization or a low utilization; only few segments have an average utilization.

LFS was first implemented in the Sprite Operating System [Ousterhout et al., 1988]: Sprite-LFS [Rosenblum and Ousterhout, 1991, 1992]. Later, Seltzer et al. [1993] implemented LFS in the BSD operating system and made some changes to its design: BSD-LFS. Unfortunately, since the launch of LFS, experiments have shown that the overall performance of LFS may seriously degrade due to the overhead of cleaning [Seltzer et al., 1993, 1995].

Subsequently, much research has been done in trying to overcome the cleaning problem. For example, Blackwell et al. [1995] have analyzed trace data from live file systems and have derived heuristics that allows the cleaner to run without interfering with normal file accesses. Since disk traffic of workstation environments is bursty [Baker et al., 1991; Ousterhout et al., 1985], cleaning can be done in the idle times in between bursts. Matthews et al. [1997] have designed adaptive algorithms to enable LFS to provide high performance across a wider range of workloads. They propose to choose a segment size that matches disk and workload characteristics, to change the cleaning policy depending on the overall disk utilization, to lower cleaning costs by using cached data during cleaning, and to reorganize data on disk to match read patterns.

In another effort to decrease the overhead of cleaning, Wang and Hu [2002] have designed a new write scheme for LFS, called WOLF (reordering Write buffer Of Log-structured File system). In this scheme, modified data are sorted in main-memory buffers into *active* and *inactive* data before they are written to disk. Consequently, in WOLF segments containing active data and segments containing inactive data are written to disk separately, which results in a bimodal distribution of data on disk: active vs. inactive. The difference with previous cleaning algorithms is that the separation is made before data are written to disk. Consequently, the amount of cleaner work is reduced. Unfortunately, separating data in main memory changes the order in which data are written to disk. Therefore, in order to support correct recovery, information on the original arrival order of data are written to disk in summary blocks together with the segments.

Similar to LD, recovery in LFS consists of restoring a previously made checkpoint and replaying the log segments written after that checkpoint. Furthermore, since LFS avoids

in-place updates as it writes in a log, LFS can quickly recover both user data and metadata to a consistent state. BSD-LFS also has the program `fsck` to check and, when necessary repair, the file system structure to survive media failures.

Several differences exist between LFS and LD. First of all, even though LD and LFS both write data segmentwise, LD is log-based rather than log-structured. In LFS, the whole disk is a log, whereas in LD, only part of the disk is a log, but the majority of the disk is used to store data in a traditional manner. Consequently, LD's design allows for the optimization of its read performance by clustering data on disk, whereas LFS's design optimizes write performance only. Second, LD provides the ability to group arbitrary commands into larger atomic units, whereas LFS does not. With grouping into ARUs, a file system on top of LD can make any change to the data on disk atomic, which increases data integrity significantly. Third, unlike LFS, LD is not a file system, but only offers low-level storage management. LD offers a more general solution to storage, and forms a basis on which complete file systems can be built. However, LD does offer disk files and clusters, which resemble somewhat files and directories in file systems.

There are also similarities between LD and LFS. As an example, both LD and LFS need some sort of cleaner process. Unfortunately, like in LFS, cleaning and reorganizing in LD may also have a negative effect on performance. Chapter 8 has discussed LD's cleaning and reorganizer strategies in great detail. Arguments have been presented to determine the possible impact and effectiveness of cleaning and reorganizing. Further research, however, is necessary to determine how LD's cleaners and reorganizers perform in a real system.

10.5 Loge and Mime

Mime [Chao et al., 1992] is a parallel storage architecture for a disk subsystem. However, as we already indicated in the beginning of this chapter, we are less interested in the parallel disk architecture of Mime, but more in the data integrity guarantees that Mime provides. Therefore, in the following discussion of Mime, we will mainly focus on its data integrity guarantees. The functionality of Mime regarding data integrity resembles the functionality in LD. Mime builds on previous work done for Loge [English and Stepanov, 1992]. We shall, therefore, first discuss Loge in more detail.

Loge

Loge is an intelligent disk controller that decides where to place blocks on disk autonomously. In the interface, the only supported data abstraction is the data block. Loge divides the disk into segments, which are totally different from LD's segments. A Loge-segment is a single data block together with some meta-information stored in sector headers of the disk, which are invisible to the host computer and which require some hardware support. A segment in LD, consists of multiple data blocks stored contiguously on disk. To avoid confusion, we will use the word Loge-segment to indicate a segment in Loge.

When Loge writes data to disk, it simply writes the blocks to the free Loge-segments closest to the current position of the disk head. Loge reserves a small portion of the disk (3% - 5%) as free segments, spread across the surface of the disk. By spreading the free

Loge-segments uniformly across the disk, free Loge-segments are always near the head of the disk, resulting in good write performance. Comparable to the Mapping in LD, Loge also keeps an indirection table which stores where blocks have been written for future access.

To enable crash recovery, the meta-information written together with each data block includes the logical block address of each block. Recovery consists of scanning the entire disk and rebuilding the indirection table with the help of the meta-information found on disk. Furthermore, because blocks are not overwritten (no in-place updates), Loge guarantees that single-request updates are atomic.

Since Loge uses a level of indirection, it can freely relocate data on disk without consequences for its users (location transparency). For example, Loge could add adaptive block rearrangement techniques [Akyürek and Salem, 1995] to improve read performance. To ensure sufficient write performance, Loge also needs to rearrange the data such that the free Loge-segments are spread sufficiently. The authors suggest that such on-line reorganizations may be done during idle periods.

Mime

As mentioned before, Mime builds on the research done for Loge. Mime still basically offers the same interface as a disk, but extends it with operations to support multiple views of data. Typically, Mime consists of a central controller (*deck*) that connects multiple disks (*cards*). Each card uses the Loge method of writing data to disk. The deck maintains a *primary index* in volatile memory, which maps logical block numbers to a (card_nr, segment_nr) pair, which denotes the physical location of the corresponding block. With this primary index, Mime supports location transparency, which allows it to rearrange data blocks transparently.

Mime supports read and write operations on blocks. Both the write of a single block and the write of a range of blocks are atomic. Furthermore, Mime also supports *visibility groups*. Each operation can be labeled as belonging to a visibility group. Visibility groups are explicitly created with a *new group* operation. Operations within a visibility group are *provisional*, because their effects are only visible within the visibility group. A special *finish* operation makes the provisional writes within a visibility group atomically visible to others. Likewise, all provisional writes can be rolled back by an *abort* operation. Operations sent outside a visibility group are *permanent* operations and are not undoable.

A finish operation does not guarantee permanence. To control the durability of data, Mime provides two operations: barrier and sync. The *barrier* operation is called within a visibility group and guarantees that after a crash the visibility group will recover to a barrier point, but not necessarily the latest barrier. The *sync* operation guarantees that all permanent operations (including a finish call) sent before the sync will survive a crash. Within a visibility group this guarantee only extends to operations protected by (i.e., issued before) a barrier.

Similar to LD, recovery in Mime consists of two phases. First, Mime recovers a checkpoint, and then, Mime replays the operations following the checkpoint. To find out which operations were done after the last checkpoint, Mime uses the meta-information stored with each Mime-segment. In addition, Mime keeps an *operations log* that contains

all synchronization and visibility operations (such as finish and barrier operations) since the last checkpoint. The log is written to disk during a checkpoint and a sync. The internal data structures that are checkpointed include the primary index and the *free list*, which keeps track of the free Mime-segments on the disks.

The data integrity guarantees provided by Mime are similar to the ones provided by LD. Both systems avoid in-place updates (shadowing), and Mime's visibility groups are comparable to LD's ARUs. Both abstractions provide atomicity with respect to recovery for a group of operations, and the changes are only local until a finish or commit, respectively. Although, Mime can recover part of a visibility group with the help of barriers, while an ARU in LD can only recover as a whole or not at all. Full transaction semantics are not supported in Mime nor in LD, since both visibility groups and ARUs do not provide serializability. That is left to higher-level software layers. Furthermore, both Mime and LD provide *monotonicity*, which means that the system guarantees that after recovery a prefix of executed commands is recovered.

A difference between Mime and LD is that Mime only offers a basic disk block interface, whereas LD offers abstractions to indicate logical and physical relationships between blocks: disk files and disk clusters. These abstractions enable LD to rearrange the blocks on disk according to the wishes of LD's clients in order to improve read performance, as presented in Chapter 8. Mime can only use heuristics when it tries to rearrange blocks on disk, as all Mime sees is a stream of data blocks.

10.6 Disk Caching Disk

The Disk Caching Disk (DCD) [Hu and Yang, 1996; Nightingale et al., 1999] uses a technique that optimizes the write performance of systems by adding a small log disk (cache disk) as a secondary disk cache. Similar to Loge, DCD offers only a basic block interface and works underneath the file system on the device or driver level. Additionally, DCD is completely transparent to a file system on top of it. Therefore, file systems can use DCD without the need for any changes.

Data written by the file system are first cached in RAM, and as soon as sufficient data has accumulated, they are written to the *cache disk* via an efficient, sequential write. At a later point in time, the data is written from the cache disk to the *normal data disk* during idle time. This process is called *destaging*. The cache disk, which has a size of one to tens of MBs, can be an actual physical disk or a reserved part of the normal data disk.

The advantage of using the cache disk is that writes to the cache disk are fast, as it is an append-only log (see also LFS). In fact, the cache disk logically forms a large and fast write cache that still provides safety against crashes. In addition, to increase performance further, DCD writes large consecutive data writes (i.e., 64 KB or more) directly to the data disk, instead of to the cache disk. This behavior is similar to LD's use of direct segments.

Since DCD is underneath a file system, it only has the physical addresses of blocks to identify blocks. Therefore, since DCD temporarily holds newly written data in the cache disk, it needs a mapping that maps physical block addresses (as used by the file system and the normal data disk) to log block addresses (as used by the cache disk). To ensure recovery of the mapping after a crash each log write is accompanied by a summary sector,

which contains changes to the mapping table. After a crash, all summary sectors on the small cache disk are scanned, and the mapping is reconstructed from the information stored in these summary sectors.

The destaging process moves data blocks written to the cache disk to their original locations in the data disk. The original locations of blocks in the cache disk are stored in the mapping. Destaging is done during idle periods, and the process is interruptible. DCD uses a last-write-first-destage algorithm in which the latest-written log segment is destaged first. The advantage of this scheme is that the disk head is always close to the head of the log. Consequently, when the destaging process is interrupted, the disk head does not have to travel far when a new log segment has to be written.

Unfortunately, the last-write-first-destage algorithm does complicate recovery, because cleaning log segments out-of-order means that during recovery, segments cannot be replayed in the exact order they were written, as there may not be a contiguous history of written segments available. Gaps may exist in the history of executed operations as log segments may have already been overwritten. In contrast, even though LD's cleaners also clean log segments out-of-order, LD keeps the log segments intact so that they can be replayed in order after a crash. Only after a checkpoint has been made are the log segments free to be reused (see Chapter 8).

Crash recovery in DCD consists of reading the entire cache disk to find the segment summaries of written log segments in order to rebuild the in-memory data structures, such as the mapping. Subsequently, a file system on top of DCD can perform its normal crash recovery activities. In other words, a file system on top of DCD still needs other techniques to recover to a consistent state. DCD only recovers its own internal data structures, but does not guarantee recovery of a consistent client data state. In contrast, LD does support recovery of a consistent client data and consistent metadata state. LD supports grouping multiple data writes into larger atomic actions, which enables file systems to go from one consistent state to the next in one atomic action. In short, LD can offer more data integrity guarantees than DCD.

Both DCD and LD can be considered log-based and not log-structured, as they both use the log only for temporary storage. However, the difference between the two is that LD uses logical block addresses in its interface, whereas DCD uses physical block addresses. As a result, LD can rearrange data blocks in the storage area to optimize read performance, whereas DCD cannot. DCD leaves clustering of data blocks on disk to the file system.

10.7 Summary

In this chapter, we looked at related work in the field of storage management. In the first part of this chapter, the discussion focused on the three problem areas modularity, data integrity, and performance. For each area we discussed related techniques used in other systems and compared them to techniques used in LD.

The main observations that can be made from comparing LD to other systems in the first three sections are:

- LD's modularization which separates file management from disk management is not as general as the modularization in other systems, such as file-system stacking

or the exokernel framework. This observation is in line with the fact that the aim of LD was not to provide a general framework, but only to allow the creation of a low-level data storage system with desirable properties such as improved data integrity.

- LD's data integrity guarantees are stronger than the guarantees offered by most disk-oriented systems. LD's ARUs enable clients to group arbitrary commands into atomic units of work, which provides client data and metadata consistency guarantees. Most file systems only guarantee metadata consistency (i.e., file-system structure integrity) and do not support arbitrary grouping of commands.

- LD's log-based architecture allows it to improve performance, as well as improve data integrity. Collective writes into the log improve write performance, clustering data in the storage area according to the client's wishes improves read performance, and log-tuples improve data integrity. Other log-based systems use the log only for metadata integrity (journaling).

- Many performance-enhancing techniques, such as explicit grouping, disk scheduling, prefetching, and changing cache replacement policies can be used in addition to the techniques currently used in LD.

In the second part of this chapter, we discussed three systems in detail: LFS, Loge/Mime, and DCD. Table 10.1 summarizes the comparison of these three systems and LD on some major issues.

Table 10.1: Summary comparing four systems.

System	Type of storage	Data integrity guarantees	Write method	Cleaner purpose	Data clustering
LD	low level storage	client & metadata	collective writes	prune log	yes
LFS	file system	limited client† & metadata	collective writes	prune log	no†
Loge/ Mime	low level storage	client & metadata	closest free segment	spread free space	no†
DCD	low level storage	metadata only	collective writes	prune log (destaging)	no

†See accompanying text

The table shows that LFS guarantees a limited form of client data integrity. We have called it limited compared to the client data integrity guarantees of LD and Mime because LFS only supports the atomicity of individual file-system calls. In contrast, the ARUs and visibility groups of LD and Mime, respectively, provide a more general method that can guarantee client data integrity across multiple calls.

The last column in the table summarizes whether the system supports data clustering to improve read performance. The original design of LFS did not include support for reorganizing data on disk. The only purpose of LFS's cleaner was to prune the log to create free space to write new segments. Later improvements on LFS [Matthews et al., 1997], however, have proposed data clustering to improve read performance. The possibility of reorganizing data on disk to improve read performance in Mime is mentioned by the authors. Unfortunately, they do not elaborate on this option much further. Implementing this option seems more difficult to do than with, for example, LD, since no information about the required clustering is present in the case of Mime.

Chapter 11

Summary and Conclusions

In this final chapter, we conclude this dissertation with a summary, some conclusions, and a look at possible future work. The main research question examined in this dissertation was whether it is possible to improve the modularity and the data integrity guarantees of disk storage systems, while still offering competitive performance. To answer this question we designed and built the Logical Disk, as described in Chapters 3 – 8. In this chapter, we examine whether we can answer our research question affirmatively.

This chapter is structured as follows. Section 11.1 starts with a summary of the previous chapters in this dissertation. Section 11.2 presents the conclusions of the research, and focuses on the question how well LD has reached its goals. Finally, in Section 11.3 we discuss some main issues that are still open, some of which have become apparent during our research of LD.

11.1 Summary of this Dissertation

Chapter 1 presented a brief overview of the subject of this dissertation: the use of disk-based storage systems. The chapter argued the importance of hard disks as the dominant medium for data storage. Unfortunately, the current use of hard disks has some problems. First and foremost, if used carelessly, the current use of disks may cause data integrity problems when confronted with system failures. For example, a system failure may cause the loss of user data because they are usually left unprotected. Techniques to avoid data integrity problems often come at the cost of substantial performance loss, and therefore, some storage systems, in particular some file systems, only protect metadata. Second, since the disk is one of the slower hardware components in a computer, it can easily become the bottleneck that limits the overall performance of computers. Third and last, the software using disks, such as DBMSs and file systems, have become increasingly complex since they incorporate complicated algorithms to manage disk storage. Correspondingly, the task of developing and implementing new DBMSs and file systems has become increasingly difficult. The chapter ends with the introduction of the Logical Disk, which has been designed to alleviate the mentioned problems.

Chapter 2 analyzed the problems with current disk usage in more detail. The main

culprits for the data integrity problems were identified: in-place updates, command reordering, and the lack for atomic multiblock updates. The performance problem is caused by a low disk bandwidth utilization, which results from the transfer (both reading and writing) of small amounts of data and from synchronous writes. In order to come to a general solution that would solve all identified problems, nine requirements for a solution were presented.

Chapter 3 presented LD, its goals, its main abstractions, and its external interface. LD forms a layer between the storage system and the disk itself. This layer provides a disk abstraction, which takes away the task of disk management from the storage system built on top of LD. The result is that a file system or DBMS can concentrate on higher-level tasks, and therefore, will be simpler to design, which increases the maintainability and extensibility of the overall computer system.

The goals of LD were summarized as follows:

(1) Improve the modularity of storage management software,

(2) Improve the functionality of disk storage systems by improving in particular their data integrity properties, and still

(3) Provide performance competitive to other systems.

Furthermore, this chapter discussed the main abstractions of LD: logical blocks, logical block addressing, disk files and disk clusters, streams, and atomic recovery units. The first three items on this list enable LD to support location transparency: clients of LD do not and need not know where the data are stored on disk. However, clients can still indicate which blocks should be clustered in order to achieve good read performance. Streams and the atomic recovery units allow LD to provide well-defined data integrity guarantees, which cover both client data as well as LD's own metadata.

The internal design of LD was discussed in Chapters 4 – 8. A general overview of how LD works was given in Chapter 4. This chapter introduced the four major types of data that LD distinguishes: client data, metadata, log data, and checkpoint data. Each type of data is stored in its own area on disk. In a nutshell, client data are stored in the storage area, where they are stored according to the clustering wishes of clients. LD avoids in-place updates, and therefore, LD always writes data to new locations on disk. Data are either written into the log first using large sequential writes and moved into the storage area at a later time, or they are directly written into the storage area via direct segments (see Chapter 5). In either way, data on disk are never overwritten directly. Data in the log are only temporarily stored there. At a later point in time, the data are moved from the log to the storage area by log cleaner processes. Whenever client data are written to disk, log tuples are written into the log that describe those client data. These log tuples are used to recover to a consistent state after a crash.

LD's metadata consists of data structures that LD uses to keep track of where client data are stored on disk. A checkpoint represents a consistent snapshot of the metadata of LD, which is used as a starting point during the recovery process after a crash. LD can subsequently recover to a more recent and consistent state by replaying the log tuples in the log, which describe the history of changes made after the last successfully made checkpoint.

Chapter 5 discussed the log in great detail. The purpose of the log is threefold. First, the log enables LD to recover to a recent and consistent state as it keeps a history of executed client commands in the form of log tuples. Second, it helps to prevent in-place updates as client data that are not written via direct segments are first written into the log. Third, it is used to improve the disk bandwidth utilization as client data are written into the log via large sequential writes. In addition, the chapter introduced direct segments, which LD uses for large sequential writes that bypass the log to improve performance. Rules were presented that indicate when direct segments could be used.

LD's data structures (i.e., metadata) were the subject of Chapter 6. The two main data structures of LD are the Mapping and the FreeMap. The former keeps a mapping between logical block addresses to physical block addresses; the latter administrates which physical blocks on disk are in use and which are free. Additional data structures are the Meta Mapping and the Root Mapping, which are used to keep track of the physical blocks that store the contents of the Mapping and FreeMap. The Mapping is implemented as a W-tree, a B-tree variant, and uses compression techniques to store the mapping information compactly. In addition, two techniques were presented that help make updates to LD's metadata efficient: the differential technique and the staccato write. The first technique makes updates to LD's metadata efficient by accumulating multiple small updates and applying them in groups. The second technique makes physically writing metadata blocks into the metadata area very efficient.

The crash recovery process of LD was discussed in Chapter 7. The purpose of the recovery process is to bring LD back to a recent and consistent state, after a crash. The chapter defined four different levels of consistency, and explained how LD recovers to the highest level of consistency: recovery consistency. A main focus of the chapter was the checkpoint. LD's checkpoint represents a snapshot of a consistent state of LD's metadata. This state is restored after a crash, and subsequently, LD replays the log tuples in the log to recover to a state that is recovery consistent.

Chapter 8 discussed the storage area of LD. It presented a novel disk layout, and the algorithms of the cleaner and reorganizer processes that maintain that layout on disk. The storage area is divided into the CDL and CDS areas. The CDL area holds data that form large consecutive ranges, such as data written within direct segments. The CDS area holds the rest of the client data. The address-slot table is used to maintain the clustering of blocks within the CDS area. The available free disk space is dynamically spread over the CDL, CDS, and metadata areas.

There are two types of processes that move data blocks on disk to maintain and improve the physical clustering of blocks: cleaners and reorganizers. Cleaners move data from the log into the storage area; reorganizers move data within the storage area and move blocks when a resize of the CDL, CDS, and metadata areas is necessary. Within the cleaner processes we distinguish the discretionary cleaner and the mandatory cleaner. The former has freedom in choosing which blocks to move from the log into the storage area, the latter has not. The chapter also identifies seven tasks that the reorganizers have to perform.

Chapter 9 described the performance measurements that were performed on a prototype implementation of LD and LDFS, a file system on top of LD. The performance of the (LD + LDFS) combination was compared to the performance of a number of other

contemporary file systems. The measurements focused on disk accesses only by using disk traces, and thereby removed the overhead of CPU activity from the measurement results. We deliberately focused on disk accesses as the disk is the slowest component in the computer for our measurements. Moreover, the speed of CPUs increases faster than the speed of hard disks, so that in the future disks will remain the relatively slowest component. Another reason to use this testing method was the lack of a proper implementation of (LD + LDFS) in an operating system kernel. The results of the measurements showed that the current prototype of LD has some strengths and weaknesses, and that overall its performance already seems to be competitive to the other tested file systems. Given that LD is only at the beginning of its development, the performance of LD may be seen as promising.

Chapter 10 presented related work and compared it to LD. The discussion of related work covers the three problem areas that LD focuses on: modularity, data integrity, and performance. The chapter gave an overview of some related research performed in each of these areas. A more detailed discussion was given of three related storage systems: LFS, Loge/Mime, and DCD.

11.2 Conclusions

This dissertation presented, discussed, and evaluated the main ideas behind the Logical Disk. The goal of the research presented in this dissertation was to examine whether it was possible to improve the modularity and the data integrity guarantees offered by disk storage systems, while still providing competitive performance. To answer this question we analyzed the problems with current disk usage, proposed some solutions, and deduced a number of requirements in Chapter 2 that must be met to solve these problems.

In order to show how these requirements can be met, we designed and built LD. The design of LD meets all of these requirements by using techniques such as direct segments, atomic recovery units, the staccato write, and a new data layout, as we explained in Chapters 3 – 8. As a result, software using LD as the underlying disk storage, such as LDFS, shows improved modularity and can easily implement data integrity guarantees (Section 9.3).

Initial performance measurements on LD and other file systems in Chapter 9 show promising results indicating that the performance of software using LD can be competitive to other storage systems. Due to the improved data integrity guarantees that LD provides, it was reasonable to expect that LD would perform a little less than other file systems providing less data integrity guarantees. Nevertheless, LD outperforms some of those file systems, and sometimes, even manages to outperform them all. Unfortunately, the final verdict on the question whether an LD-like approach will provide competitive performance to other storage systems in real-life situations is still unclear. Further research, implementation effort, and performance measurements are necessary to determine a more conclusive answer. However, the improved modularity and especially the much improved data integrity guarantees alone already make LD worthwhile.

In conclusion, we believe that it is indeed possible to improve the modularity of disk storage systems, and to improve their data integrity guarantees, while still providing com-

petitive performance. We base our claim on the experiences we obtained after designing, building, and performing quite some measurements on the Logical Disk.

During the design, implementation, and the measurements of LD, we have also made the following observations:

- *LD's overall performance is sensitive to LD's metadata performance.*
  The results of LD's performance measurements in Chapter 9 taught us that, in the current prototype, the read performance of metadata has a strong influence on the overall performance. More precisely, the clustering of metadata is important for LD's metadata read performance. The current write performance of LD's metadata is good, due to the staccato method which is used to write metadata in the metadata area (see Chapter 6). Increasing the read performance of metadata, however, could raise LD's overall performance in some performance measurements significantly.

  In our design of LD, we assumed that clustering of metadata would not be very important since applications generally do not access metadata sequentially in large amounts. Unfortunately, in our performance measurements LD's implementation did read metadata sequentially in large amounts. However, these sequential accesses did not only originate from the performance experiments, but from LD's internal cleaner and reorganizer algorithms as well. Therefore, improving LD's cleaner and reorganizer algorithms to lower the number of sequential accesses to LD's metadata may improve LD's overall performance. In addition, other solutions to increase LD's read performance of metadata are possible. These solutions were discussed Section 9.7.

- *The amount of reorganizing work seems acceptable.*
  One reason why we cannot give a final verdict on LD's overall performance is because the prototype of LD lacks fully functional reorganizer processes. However, in Section 9.6, we looked at the amount of reorganization overhead in our performance measurements of Chapter 9. The conclusion was that the amount of overhead is relatively small. It is already possible to reorganize once per day without disturbing any users if the disk is idle for only 2-3% per day. Therefore, even though experiments with fully functional reorganizers still need to be done, we are confident that, in practice, reorganizing can be done in the background.

- *ARUs are a simple mechanism for programmers to achieve data integrity.*
  The experience of using ARUs in the implementation of LDFS taught us that ARUs are an effective and simple programming abstraction to implement data integrity guarantees. In LDFS, each logical operation is protected against corruption because it is executed as an atomic update, even if the logical operation required updates to multiple data structures on disk. All that was required in the implementation of LDFS was to put each logical operation within an ARU. The relatively small size of the source code of LDFS (see Section 9.3) is partly due to the effectiveness of LD's ARU mechanism. The success of ARUs in the implementation of a file system suggests that their application in the implementation of DBMSs may be successful as well. Moreover, since DBMSs put greater emphasis on data integrity than file

systems do, the positive effect of ARUs on the implementation of a DBMS may be even larger than it already was on LDFS.

- *Direct segments are an effective technique to improve the performance of user data logging systems.*
  Using traditional logging to protect user data against system failures results in performance loss since data has to be written twice. However, with direct segments, LD can at least partly avoid writing data twice. The performance measurements show that direct segments in combination with LD's disk layout (i.e., dividing the storage area in the CDS and CDL areas) can be very effective in raising LDFS's write performance. These performance results suggest that it may be worthwhile to investigate whether lowering the size of direct segments from 256 KB to 128 KB or even 64 KB can yield even better overall results. Perhaps smaller direct segments may be beneficial in combination with reorganizing (i.e., clustering) the data in the CDL area.

- *Efficiently supporting disk block sizes as small as 512 bytes is possible.*
  File systems normally support only relatively large block sizes of 4 or 8 KB in order to increase performance without suffering too much internal fragmentation. However, performance measurements on LDFS suggest that a block size as small as 512 bytes can also be efficiently supported. This success is possible because LD transparently clusters data blocks on disk and because LD is able to store its block administration compactly in its Mapping.

11.3 Future Work

In this section, we present a number of issues that may be addressed in the future. Some of the issues have already been mentioned in previous chapters. The identified future work can be divided into two categories. The first category, which covers the first three issues mentioned below, deals with changes, additions, and enhancements on LD's current design. The second category, which covers the last two issues mentioned below, concerns the prototype of LD and its application in real life.

- *Separate the logical and physical interfile clustering mechanisms.*
  The current design of LD uses the disk cluster to indicate a logical relationship between disk files (i.e., these disk files can be referred to as a single unit). In addition, the disk cluster is also used to indicate a physical relationship between those disk files (i.e., these disk files should be stored clustered on disk). In other words, the current design of LD uses the same mechanism to express both a logical and a physical relationship between disk files.

  A consequence of using the same mechanism in the current addressing scheme is that a disk file's cluster_id determines with which other disk files it is physically clustered. In other words, it is not possible to change the interfile clustering property of a disk file without changing its logical cluster_id, which is not a desirable property. Therefore, as we already mentioned in Section 3.3, in the future, LD

should support a more general clustering mechanism that allows any two disk files to be clustered, not only disk files that belong to the same disk cluster. The plan is to let the disk cluster indicate only a logical relationship; the physical relationship is indicated differently. A separate data structure must be developed to keep track of the desired interfile clustering.

- *Improve metadata performance.*
  As we explained above, LD's read performance of metadata has a significant influence on LD's overall performance. Section 9.7 already presented possible solutions: improve LD's cleaner and reorganizer algorithms, increase the metadata cache size, increase the metadata block size, and introduce metadata reorganizers. Currently, work is in progress to implement some of these solutions and evaluate their impact.

- *Investigate the possibility of extending ARUs to block-level transactions.*
  Chapter 2 introduced the possibility of block-level transactions in a disk storage system. The ARU does not provide the same properties as a transaction because the ARU does not provide full isolation and supports only limited durability. It is tempting to extend ARUs with the property of full isolation, or even to extend ARUs to full-fledged block-level transactions. However, more research is necessary to examine the usefulness of block-level transactions on such a low level. Some preliminary research suggests that the granularity of blocks is too coarse for clients to use LD's transactions effectively. Often clients need to update data items that are smaller than blocks, which means that clients still need to implement their own locking and some concurrency control, at least, if LD only offers block-level transactions. In that case, the question rises what the advantage is of LD supporting block-level transactions? This question must be addressed in future research. One interesting possibility to consider is a version of LD supporting byte-level operations and ARUs with full isolation.

- *Implement an in-kernel version of LD.*
  In order to evaluate LD in a more realistic, production-like environment, we need an in-kernel implementation of LD. Currently, an integration of LD into the FreeBSD kernel is in progress. A complete in-kernel implementation also includes cleaner and reorganizer processes that can run concurrently in the background. With such a complete implementation, LD could be compared head-to-head to other file systems.

- *Evaluate other applications on top of LD.*
  Currently, only a simple file system exists on top of LD. In the future, more sophisticated storage systems, such as different types of DBMSs, should be implemented to evaluate LD's suitability for storage systems. Furthermore, this would yield more information on how well LD increases the modularity of software.

Appendix A

Performance Results II

In Chapter 9, we performed some performance measurements. The performance results were influenced by how the disk was configured: with write caching enabled and tagged command queuing with simple tags, or with write caching disabled and tagged command queuing with ordered tags. The former configuration yielded performance results that were potentially too optimistic, whereas the latter configuration yielded results that were too conservative. Fortunately, the results provide sufficient indication to determine whether LD has performance competitive to other file systems

For clarity, Chapter 9 presented only the performance results of the test runs with write caching enabled and tagged command queuing with simple tags. In this appendix, we present the results obtained from performing the test runs with write caching disabled and tagged command queuing with ordered tags only. For a description of the experiments performed, see Section 9.4. To save space, we present only the results of create, write, read, and delete test-phases.

The performance numbers we present in this section are all lower than the numbers presented in the Section 9.5, on page 254. The performance losses are largely due to the inability of the disk to write back-to-back (see Section 9.4.8).

All the performance numbers are the average of running the disk traces three times through our trace driver. In over 99% of the cases, the variations in the performance numbers measured in these three runs were within 5%. Deviations up to 10% were also present, but only in cases for which the total absolute time was mostly in the order of only one second.

A.1 Create Test-Phase

With write caching disabled and tagged command queuing with ordered tags, the create performance of all tested file systems decreases. The results are shown in Table A.1, a graphic representation of the results is shown above the table. The relative create performance of all file systems remains almost the same compared to the results presented in Section 9.5. Most file systems lose around 30% in performance. However, XFS and Soft lose the most — up to 60% of their throughputs. The performance of FFS decreases only

marginally because the overhead of the synchronous metadata writes still dominates the performance.

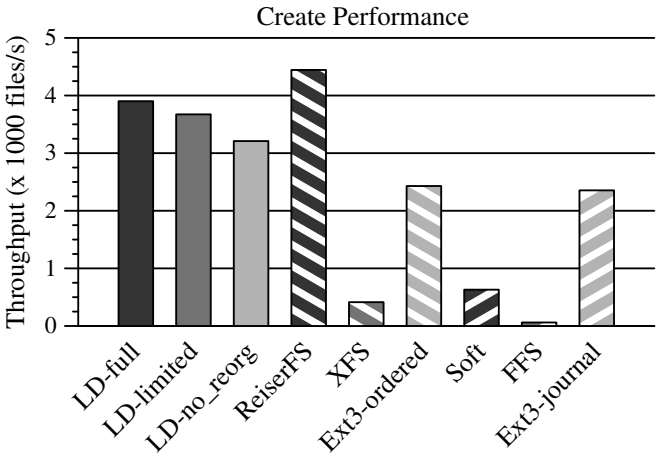


Table A.1: Create Performance. Results of the create test-phase for 50,000 files. The disk had write-caching disabled and used tagged command queuing with ordered tags.

File system	Create throughput (files/sec)	% of perf. LD-full
LD-full	3,901	100
LD-limited	3,673	94.2
LD-no_reorg	3,209	82.3
ReiserFS	4,444	113.9
XFS	414	10.6
Ext3-ordered	2,429	62.3
Soft	630	16.1
FFS	61	1.6
Ext3-journal	2,354	60.3

In conclusion, in this measurement, LD's create performance shows very good create performance, that is certainly competitive to the create performance of other systems.

A.2 Write Test-Phase

Figure A.1 shows the results of the write test. The top pair of graphs show the write performance of the three tested versions of LD. The shapes of the lines in the graph are

very similar to the one in Figure 9.6, on page 260. The main difference is the scale. The maximum write performance of LD-full now reaches only just over 5 MB/s, the highest write score in the test. The four dips in the graph at 16 KB, 64 KB, 160 KB, and 1 MB are due to a resize operation during the test, as explained in Section 9.5.4.

The second pair of graphs also show patterns similar to the corresponding graphs in Figure A.1. Ext3-ordered shows the same drop in performance at 64 KB due to the indirect block, even though the drop is smaller. ReiserFS shows very good write performance. XFS remains a slow starter, and ends with high write scores for large files.

Surprisingly, the graphs in this test show that both LD-full and Ext3-journal outperform FFS and Soft on file sizes up to 16 KB. The poor performance of the latter two file systems, however, is mainly due to the inability of the disk to write back-to-back (see Section 9.4.8). In the performance numbers presented in Section 9.5 the back-to-back problem is not present, and there FFS and Soft outperform LD-full and Ext3-journal. Since both LD-full and Ext3-journal write data twice, it is to be expected that their performance will be lower than the performance of file systems that do not log data or log only metadata.

Similar to what we have done in Section 9.5, we have calculated the weighted average write performance of each file system. The results are shown in Table A.2, and a graphic representation is shown above the table. As expected, the results show that the relative write performances are similar to the results shown in Table 9.9 in Section 9.5. The only major difference is the poor performance of FFS and Soft, which is due to the back-to-back write problem as explained above.

A.3 Read Test-Phase

Since write caching and tagged command queuing have effect on write commands and not read commands, it is not surprising that the results of the read test-phase in this section are very similar to the results presented in Section 9.5.5. The results are shown in Figure A.2. One notable difference between the two read performance results is that the read performance of ReiserFS for small files is almost halved. The probably explanation for this drop is the enormous (up to 300 MB) amount of data that ReiserFS writes during these particular measurements. These writes probably suffered from the back-to-back write problem again. Table A.3 shows the weighted average read performances for each tested file system. For a more detailed discussion of the read performance measurement, we refer to Section 9.5.5.

A.4 Delete Test-Phase

Figure A.3 shows the results of the delete test. All file systems show behavior similar to the results described in the delete test of our first set of performance figures. As expected, all results have decreased somewhat. The most noticeable drop in performance is displayed by Soft, which drops below the delete performance of Ext3-journal, probably due to the back-to-back write problem again. The graphs are shown in Figure A.3. Table A.4 shows the weighted average delete performance for each tested file system, which

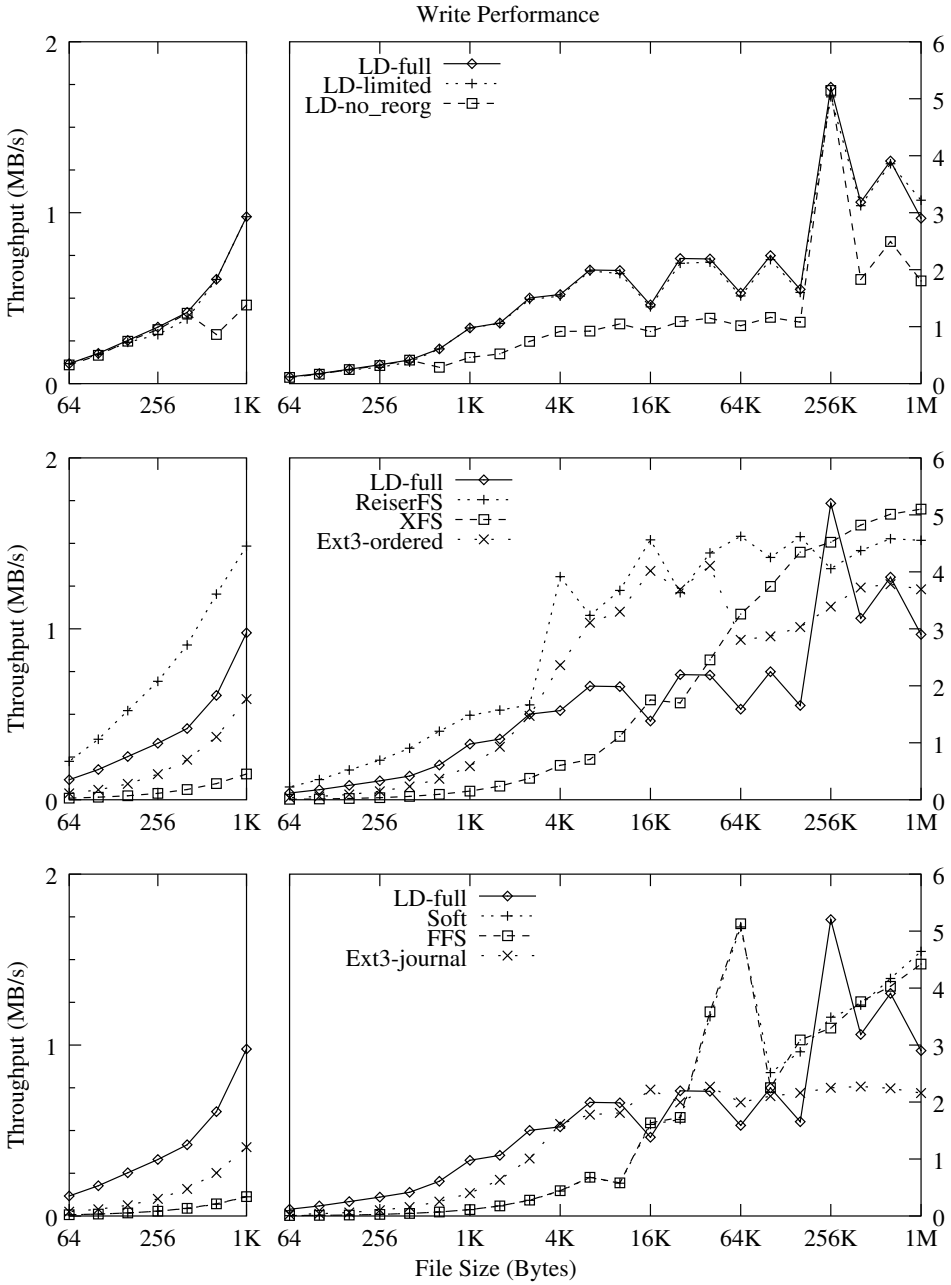


Figure A.1: Results of the write test-phase. The disk had write-caching disabled and used tagged command queuing with ordered tags.

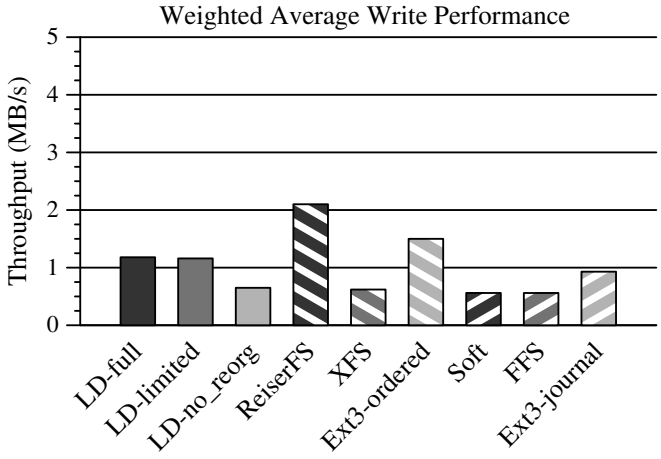


Table A.2: Weighted Average Write Performance. The disk had write-caching disabled and used tagged command queuing with ordered tags.

File system	Write throughput (MB/sec)	% of perf. LD-full
LD-full	1.18	100
LD-limited	1.16	98.1
LD-no_reorg	0.65	55.0
ReiserFS	2.10	178.1
XFS	0.62	52.9
Ext3-ordered	1.50	127.1
Soft	0.56	47.4
FFS	0.56	47.6
Ext3-journal	0.93	78.4

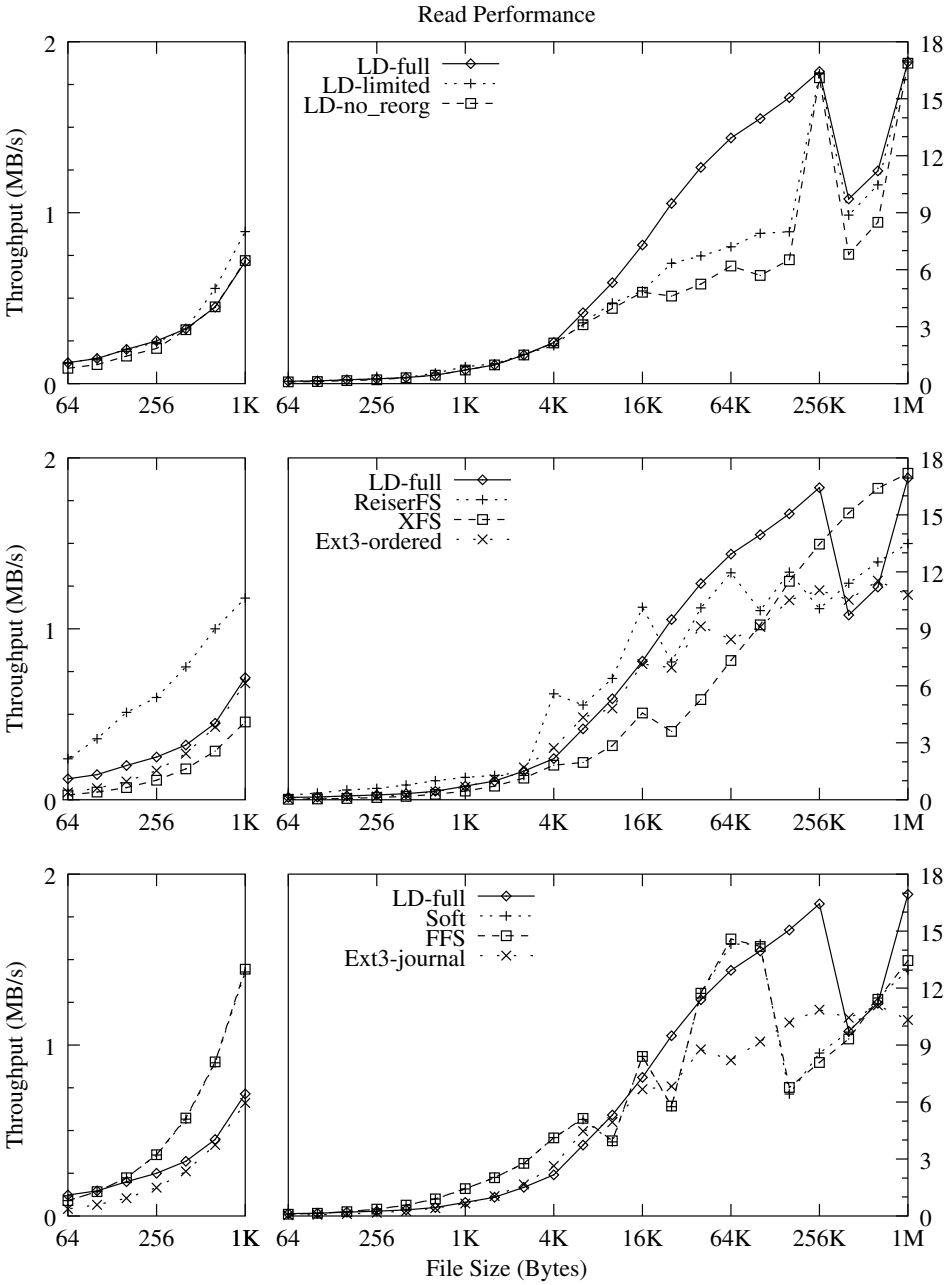


Figure A.2: Results of the read test-phase. The disk had write-caching disabled and used tagged command queuing with ordered tags.

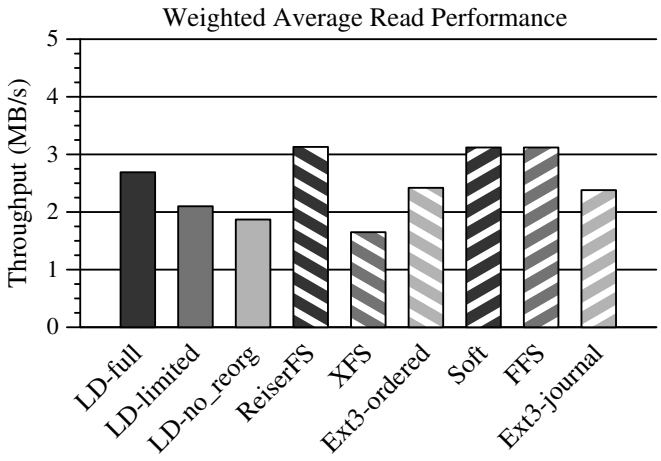


Table A.3: Weighted Average Read Performance. The disk had write-caching disabled and used tagged command queuing with ordered tags.

File system	Read throughput (MB/sec)	% of perf. LD-full
LD-full	2.69	100
LD-limited	2.10	78.2
LD-no_reorg	1.87	69.6
ReiserFS	3.13	116.7
XFS	1.65	61.6
Ext3-ordered	2.42	90.0
Soft	3.12	116.0
FFS	3.12	116.3
Ext3-journal	2.38	88.7

includes the results of the experiment in which LD has increased its metadata cache from 8 MB to 32 MB. For a more detailed discussion of the delete performance measurement, we refer to Section 9.5.6.

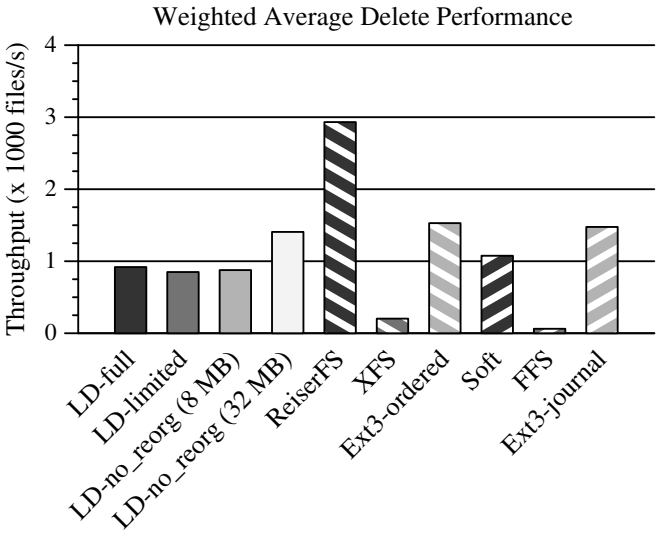


Table A.4: Weighted Average Delete Performance. The disk had write-caching disabled and used tagged command queuing with ordered tags.

File system	Delete throughput (files/sec)	% of perf. LD-full
LD-full	919	100
LD-limited	850	92.5
LD-no_reorg (8 MB)	877	95.4
LD-no_reorg (32 MB)	1,407	153.2
ReiserFS	2,932	319.1
XFS	203	22.1
Ext3-ordered	1,529	166.4
Soft	1,076	117.1
FFS	62	6.7
Ext3-journal	1,476	160.7

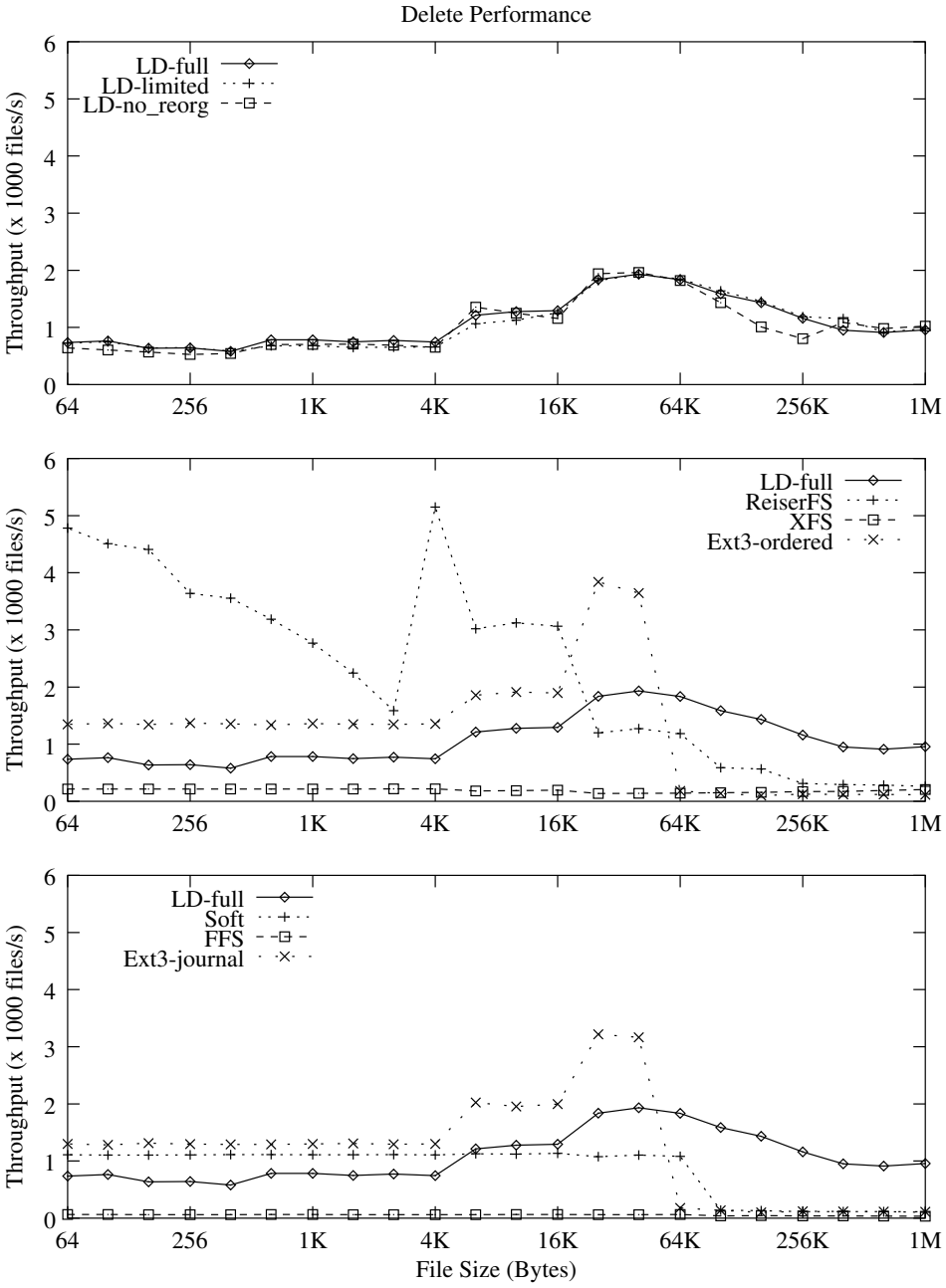


Figure A.3: Results of the delete test-phase. The disk had write-caching disabled and used tagged command queuing with ordered tags.

Samenvatting

Het Ontwerp van een Schijfbeheersubsysteem met een Hoge Integriteit

Opslag van gegevens is één van de hoofdtaken van een computer. Het is van belang dat deze taak betrouwbaar en efficiënt gebeurt. De betrouwbaarheid van gegevensopslag heeft betrekking op de integriteit van de opgeslagen gegevens. Met andere woorden, de computer moet de gegevens duurzaam en veilig opslaan. In het bijzonder, de gegevens mogen niet inconsistent raken of zelfs (deels) verloren gaan wanneer een systeemstoring optreedt. De efficiëntie van gegevensopslag heeft betrekking op prestaties: opgeslagen gegevens moeten snel benaderd kunnen worden. Aangezien computers steeds meer gegevens te verwerken krijgen is de snelheid waarmee gegevens geschreven en gelezen kunnen worden van grote invloed op de prestaties van de hele computer.

Echter, hoewel prestaties belangrijk zijn, zijn we ervan overtuigd dat de integriteit van gegevensopslag nog belangrijker is. Mensen vertrouwen hun waardevolle gegevens toe aan de computer en de computer is dan ook verantwoordelijk voor het bewaken van de integriteit van de opslag van die gegevens. We pleiten er dan ook voor dat de kwaliteitsaspecten van gegevensopslag, zoals de integriteit van gegevensopslag, meer aandacht krijgen en verbeterd worden. Helaas wordt deze mening te weinig gedeeld door degene die computers bouwen en programmatuur voor computers schrijven. Maar al te vaak hechten zij meer waarde aan prestaties en omdat het waarborgen van gegevensintegriteit meestal ten koste gaat van de prestaties, krijgt gegevensintegriteit dan ook vaak een lagere prioriteit.

Sinds jaar en dag is de harde schijf het meest gebruikte medium in computers dat gebruikt wordt voor het opslaan van grote hoeveelheden gegevens die direct beschikbaar moeten zijn. In dit proefschrift staat dan ook het gebruik van de harde schijf als opslagmedium in opslagsystemen centraal. Wij concentreren ons vooral op het verbeteren van de kwaliteit van het gebruik van schijfopslag. Ons onderzoek richt zich op het kleinschalig gebruik van opslagsystemen. Oftewel opslagsystemen met slechts één enkele harde schijf, zoals de thuiscomputer of de computer in een klein bedrijf. We presenteren ideeën die de modulariteit van programmatuur die gebruik maakt van harde schijven verbeteren. Daarnaast presenteren we ideeën die de manier waarop gegevens op de harde schijf worden opgeslagen verbeteren opdat de integriteit van de opgeslagen gegevens verbeterd wordt. We tonen aan dat we al deze verbeteringen kunnen verkrijgen zonder al te veel aan prestaties te verliezen. Onze ideeën kunnen worden toegepast zonder dat er aanpassingen aan de harde schijf zelf nodig zijn.

In het bijzonder concentreren wij ons op het handhaven van gegevensintegriteit na een systeemstoring, zoals bijvoorbeeld een stroomuitval. Zelfs binnen de context van een thuiscomputer kan het verlies van gegevens op schijf desastreus zijn voor de eigenaar van die gegevens. Helaas treft de gemiddelde thuisgebruiker geen voorzorgsmaatregelen, zoals het regelmatig maken van reservekopieën, om gegevensverlies door een systeemstoring te kunnen herstellen.

Er zijn allerlei soorten applicaties die een harde schijf gebruiken voor gegevensopslag. Twee veelvoorkomende applicaties zijn bestandssystemen (*file systems*) en gegevensbanksystemen (*database management systems*). Een bestandssysteem is meestal onderdeel van een besturingssysteem. Een besturingssysteem is de programmatuur die alle componenten van een computer, zoals de harde schijf, de printer, het toetsenbord, de grafische kaart, enz. samen laat werken. Het bestandssysteem is het onderdeel dat zorgt voor gegevensopslag en ondersteunt meestal abstracties zoals 'bestanden' en 'mappen' opdat gebruikers hun gegevens gestructureerd kunnen opslaan.

Een gegevensbanksysteem is een groot en complex stuk programmatuur dat speciaal ontworpen is om grote hoeveelheden gegevens op te slaan, om hun integriteit te bewaken en om (gelijktijdige) toegang tot die gegevens te verzorgen. Een gegevensbanksysteem kan geïmplementeerd worden boven op een besturingssysteem. In dat geval maakt het gegevensbanksysteem gebruik van het onderliggende bestandssysteem in het besturingssysteem voor de opslag van gegevens op een harde schijf. Maar het kan ook zo geïmplementeerd worden dat het gegevensbanksysteem het onderliggende bestandssysteem grotendeels omzeilt en zelf rechtstreeks de gegevensopslag op een harde schijf beheert. De reden om het bestandssysteem te omzeilen is vaak prestatieverbetering; de ondersteuning voor gegevensopslag die een gegevensbanksysteem nodig heeft, sluit vaak niet aan op hetgeen het onderliggende bestandssysteem levert.

De kwaliteit en prestaties van gegevensopslag wordt onder meer bepaald door de manier waarop een bestandssysteem of gegevensbanksysteem de harde schijf gebruikt. In dit proefschrift identificeren we drie soorten problemen die betrekking hebben op het huidige gebruik van harde schijven.

(1) **Gegevensintegriteitsproblemen**: De integriteit van de op harde schijf opgeslagen gegevens blijft niet altijd behouden na een systeemstoring.

(2) **Prestatieproblemen**: De harde schijf is vaak een knelpunt voor de prestaties van het computersysteem als geheel.

(3) **Modulariteitsproblemen**: De modulariteit van programmatuur die gebruik maakt van harde schijven moet verbeterd worden.

Het huidige onvoorzichtige gebruik van harde schijven brengt helaas integriteitsproblemen met zich mee wanneer systeemstoringen zich voordoen. Helaas gaan maatregelen om gegevensintegriteitsproblemen te voorkomen vaak gepaard met significante prestatieverliezen. Daarom beschermen veel opslagsystemen, in het bijzonder een aantal bestandssystemen, alleen metagegevens. De belangrijkste gegevensintegriteitsproblemen zijn: *in-place updates* (op-de-plaats veranderingen), opdrachtherordening en het gebrek aan ondersteuning voor ondeelbare multiblokveranderingen.

Omdat de harde schijf één van de langzamere onderdelen van een computer is, is het al snel een knelpunt dat de algehele prestatie van de computer kan belemmeren. De prestatieproblemen worden veroorzaakt door de lage bandbreedtebenutting van de schijf, welke het gevolg is van het oversturen (zowel 'lezen' als 'schrijven') van slechts kleine hoeveelheden gegevens en van synchrone schrijfopdrachten.

De programmatuur die harde schijven gebruikt, zoals gegevensbanksystemen en bestandssystemen, is steeds complexer geworden omdat ze ingewikkelde algoritmes bevat om de gegevens op schijf te beheren. Dienovereenkomstig is ook het ontwikkelen en implementeren van nieuwe gegevensbanksystemen en bestandssystemen steeds meer gecompliceerd geworden. Elk opslagsysteem bevat echter altijd ondersteuning voor schijfbeheer. Dit schijfbeheersubsysteem wordt voor elk opslagsysteem bijna elke keer opnieuw ontwikkeld en geïmplementeerd. Echter, omdat dit subsysteem ongeveer dezelfde functionaliteit levert aan elk opslagsysteem is het aantrekkelijk om éénmalig een algemeen en herbruikbaar schijfbeheersubsysteem te maken. Het ontwikkelen en implementeren van een nieuw opslagsysteem wordt dan eenvoudiger dankzij de herbruikbaarheid van zo'n algemeen schijfbeheersubsysteem.

Een schijfbeheersubsysteem zou kunnen zorgen voor de details van het beheren van een harde schijf, zoals schijfblokallocatie, geschikte clustering, enz. De programmatuur die gebruik maakt van dit schijfbeheersubsysteem kan zich dan concentreren op het implementeren van concepten op een hoger niveau, zoals relationele tabellen, bestanden en mappen, toegangsrechten, enz. Hierdoor wordt het maken van hogerniveau-opslagsystemen makkelijker. Bovendien kan een hogerniveau-opslagsysteem pas een goede ondersteuning bieden voor gegevensintegriteit indien het gebouwd wordt op een goede basis. Een losstaand schijfbeheersubsysteem kan zo'n basis zijn als het functionaliteit aanbiedt met een duidelijke semantiek met betrekking tot gegevensintegriteitsgaranties.

Onze ideeën voor het verbeteren van de kwaliteit van schijfgebruik presenteren wij aan de hand van een schijfbeheersubsysteem dat wij de *Logical Disk* (Logische Schijf) of kortweg LD hebben genoemd. De doelen van de *Logical Disk* zijn:

(1) Het verbeteren van de modulariteit van programmatuur die gegevensopslag op harde schijven beheren,

(2) Het verbeteren van de functionaliteit van hardeschijfopslagsystemen door met name hun eigenschappen met betrekking tot gegevensintegriteit te verbeteren, en toch

(3) Prestaties te kunnen leveren die concurreren met andere bestaande opslagsystemen.

De belangrijkste abstracties die de *Logical Disk* biedt zijn: gegevensblokken ('blokken' in het kort), logische blokadressering, *disk files* (schijfbestanden) en *disk clusters* (schijfclusters), *command streams* (opdrachtstromen) en *atomic recovery units* (ondeelbare hersteleenheden). De eerste drie begrippen in deze opsomming stellen LD in staat om locatietransparantie te ondersteunen: een klant van LD (de programmatuur die gebruik maakt van LD) weet niet waar zijn gegevens op de harde schijf worden opgeslagen. Het bijzondere van LD is dat een klant toch kan aangeven welke blokken hij dicht bij elkaar wil opslaan op de harde schijf om goede leesprestaties te houden. Met *streams* en *atomic recovery units* kan LD goed-gedefinieerde garanties voor de integriteit van de gegevens bieden. Deze garanties dekken zowel de klantgegevens als LD's eigen metagegevens.

Met *streams* kan een klant aangeven of zijn opdrachten in een specifieke volgorde moeten worden uitgevoerd vanwege integriteitseisen, of dat de volgorde juist bepaald mag worden door LD om optimale prestaties te verkrijgen. Met *atomic recovery units* biedt LD de klant de mogelijkheid om meerdere schrijfopdrachten te combineren in één ondeelbare opdracht.

Intern onderscheidt LD vier soorten gegevens die elk in zijn eigen gebied op de harde schijf wordt opgeslagen: klantgegevens worden opgeslagen in het 'opslaggebied', metagegevens in het 'metagegevensgebied', logboekgegevens in het 'logboekgebied' en *checkpoint*gegevens in het '*checkpoint*gebied'. In het opslaggebied worden klantgegevens opgeslagen rekening houdend met de clusteringwensen van klanten. LD vermijdt *in-place updates* en daarom worden gegevens altijd op nieuwe locaties op de harde schijf geschreven. Gegevens worden ófwel eerst in het logboek op de harde schijf geschreven met behulp van een lange, sequentiële schrijfactie en dan op een later moment naar het opslaggebied verplaatst, ófwel rechtstreeks in het opslaggebied geschreven via 'directe segmenten'. In beide gevallen worden de gegevens op de harde schijf nooit rechtstreeks overschreven. Daarnaast worden elke keer wanneer klantgegevens naar de harde schijf geschreven worden ook logboekvermeldingen (*log tuples*) in het logboek geschreven welke die klantgegevens beschrijven. Deze logboekvermeldingen worden gebruikt om LD in een consistente toestand te brengen na een systeemstoring.

Het doel van het logboek is drieledig. Allereerst, LD kan na een systeemstoring dankzij het logboek weer in een recente en consistente toestand gebracht worden doordat het logboek een geschiedenis van uitgevoerde klantopdrachten bevat. Ten tweede, het logboek helpt het voorkomen van *in-place updates* doordat klantgegevens die niet geschreven worden via directe segmenten eerst in het logboek geschreven worden. Ten derde, het logboek verbetert de bandbreedtebenutting van de harde schijf doordat klantgegevens met lange, sequentiële en dus efficiënte schrijfacties in het logboek geschreven worden.

Een andere manier om de prestaties te verbeteren is het gebruik van directe segmenten. Directe segmenten omzeilen het logboek en voorkomen dus een dubbele schrijfactie. Klantgegevens die met behulp van directe segmenten worden geschreven worden namelijk rechtstreeks in het opslaggebied geschreven. In dit proefschrift presenteren we regels die aangeven wanneer directe segmenten het beste gebruikt kunnen worden.

LD's metagegevens bestaan uit datastructuren die LD gebruikt om bij te houden waar de klantgegevens op de harde schijf zijn opgeslagen. De twee belangrijkste datastructuren in LD zijn de *Mapping* en de *FreeMap*. De *Mapping* houdt een afbeelding van logische blokadressen op fysieke blokadressen bij. De *FreeMap* houdt bij welke fysieke blokken op de harde schijf in gebruik zijn en welke beschikbaar zijn. De *Mapping* is geïmplementeerd met behulp van een W-boom, een B-boom variant, en gebruikt compressietechnieken om de informatie compact op te slaan. Andere datastructuren binnen LD zijn de *Meta Mapping* en de *Root Mapping*, die bijhouden waar de fysieke blokken met de inhoud van de *Mapping* en de *FreeMap* zelf op schijf staan.

LD gebruikt twee technieken om veranderingen aan LD's metagegevens efficiënt te verwerken: de differentiële techniek en staccato schrijven. De eerste techniek maakt veranderingen aan LD's metagegevens efficiënt door meerdere kleine veranderingen op te sparen en daarna in groepjes te verwerken. De tweede techniek maakt het feitelijke fysiek schrijven van blokken met metagegevens in het metagegevensgebied op schijf efficiënt.

Om na een systeemstoring een recente en consistente toestand te kunnen herstellen maakt LD regelmatig een *checkpoint*. Een *checkpoint* representeert een *snapshot* (momentopname) van de metagegevens van LD. Dit *snapshot* wordt gebruikt als startpunt bij het herstelproces na een systeemstoring. Vervolgens speelt LD de logboekvermeldingen in het logboek na om een recente en consistente toestand te herstellen. Het logboek bevat de geschiedenis van de gedane veranderingen na de laatst succesvol gemaakte *checkpoint*. De consistente toestand die zo wordt hersteld is consistent met betrekking tot zowel LD's metagegevens als de klantgegevens op de harde schijf.

In dit proefschrift presenteren we een nieuwe blokindeling op schijf en algoritmes voor 'schoonmaak' (*cleaner*) en 'reorganisatie' (*reorganizer*) processen die zorg dragen voor het efficiënte gebruik van deze blokindeling. Het opslaggebied is verdeeld in het CDL- en CDS-gebied (het grote klantgegevensgebied en het kleine klantgegevensgebied). Het CDL-gebied bevat klantgegevens die meerdere, aaneengesloten blokken bestrijken, zoals gegevens die geschreven worden in directe segmenten. Het CDS-gebied bevat de overige klantgegevens. LD verdeelt de beschikbare, vrije hardeschijfruimte dynamisch over de CDL-, CDS- en metagegevensgebieden.

Er zijn twee soorten processen die blokken op schijf verplaatsen om de fysieke clustering van blokken te onderhouden: 'schoonmaak-' en 'reorganiseer'processen. Schoonmaakprocessen verplaatsen gegevens van het logboek naar het opslaggebied; reorganiseerprocessen verplaatsen gegevens binnen het opslaggebied en verplaatsen blokken om de grootte van de CDL-, CDS- en metagegevensgebieden te veranderen. We onderscheiden tevens twee soorten schoonmaakprocessen: het facultatieve schoonmaakproces en het obligatoire schoonmaakproces. Het eerste proces heeft vrijheid in het kiezen welke blokken uit het logboek naar het opslaggebied te verplaatsen, het tweede proces heeft die vrijheid niet. Verder identificeren we in het proefschrift zeven taken die de reorganiseerprocessen moeten volbrengen.

We hebben prestatiemetingen verricht met prototypes van LD en LDFS. Dat laatste is een bestandssysteem dat gebruik maakt van LD. De prestaties van de combinatie (LD + LDFS) zijn vergeleken met de prestaties van een aantal andere eigentijdse bestandssystemen. Wij concentreerden ons op het meten van alleen schijfactiviteit, waarmee de kosten van processoractiviteit (*CPU activity*) buiten de meetresultaten gehouden zijn. De resultaten van onze metingen lieten zien dat het huidige prototype van LD een aantal sterke en zwakke punten heeft, maar zijn algehele prestatie lijkt nu al te kunnen concurreren met de prestaties van de overige geteste bestandssystemen. Aangezien we in onze metingen slechts een prototype van LD hebben getest, waaraan nog veel verbeteringen mogelijk zijn, lijkt LD veelbelovend.

Bovendien bleek dat dankzij de functionaliteit van LD, LDFS relatief eenvoudig en snel geïmplementeerd kon worden. Een vergelijking van de broncode van LDFS en de overige geteste bestandssystemen liet zien dat LDFS verreweg het kleinst is. Dit resultaat ondersteunt onze hypothese dat de modulariteit van programmatuur dat gebruik maakt van harde schijven verbeterd kan worden. Ook de resultaten van een *crash*test, waarin we expres een systeemstoring teweegbrachten, lieten zien dat zelfs met de verbeterde garanties op gegevensintegriteit LDFS binnen enkele seconden kan herstellen van een systeemstoring.

De belangrijkste bijdrage van dit proefschrift is de presentatie, bespreking en evaluatie van de ideeën achter de *Logical Disk*. Het doel van het onderzoek in dit proefschrift was om te onderzoeken of het mogelijk is de modulariteit en garanties aangaande gegevensintegriteit van een opslagsysteem te verbeteren en tóch concurrerend te kunnen presteren. Om deze vraagstelling te beantwoorden hebben we de *Logical Disk* ontwikkeld en geïmplementeerd. Op grond van metingen met een prototype en andere bestaande bestandssystemen concluderen wij dat het inderdaad mogelijk is om de modulariteit en de garanties aangaande gegevensintegriteit van een opslagsysteem te verbeteren en toch concurrerend te presteren.

In dit proefschrift wordt ook een aantal punten genoemd dat mogelijkerwijs in de toekomst kan worden aangepakt. Ten eerste, het clusteringmechanisme waarmee gebruikers kunnen aangeven welke *disk files* geclusterd moeten worden opgeslagen kan verbeterd worden. Ten tweede, de snelheid van het lezen van LD's metagegevens is van grote invloed op de algehele snelheid van LD. Wij identificeren daarom ook een aantal mogelijke oplossingen om de leesprestaties van LD's metagegevens te verbeteren. Ten derde, LD's *atomic recovery units* kunnen mogelijkerwijs uitgebreid worden tot 'bloktransacties'. Als laatste punt wordt genoemd om de implementatie van LD in een besturingssysteem te integreren en meerdere applicaties te ontwikkelen die gebruik maken van LD. Met deze applicaties kan LD uitgebreid geëvalueerd worden.

Bibliography

Akyürek S. and Salem K., 1995. Adaptive Block Rearrangement. In *ACM Transactions on Computer Systems*, 13(2):pp. 89–121.

Bach M.J., 1986. *The Design of the UNIX Operating System*. Prentice-Hall, New York, NY.

Baker M., Asami S., Deprit E., Ousterhout J., and Seltzer M., 1992. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 10–22. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society, Boston, Massachusetts.

Baker M.G., Hartman J.H., Kupfer M.D., Shirriff K.W., and Ousterhout J.K., 1991. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 198–212. Association for Computing Machinery SIGOPS.

Bayer R. and McCreight E.M., 1972. Organization and Maintenance of Large Ordered Indexes. In *Acta Informatica*, 1(3):pp. 173–189. ISSN 0001-5903 (print), 1432-0525 (electronic). Also published in/as: ACM SIGFIDET 1970, pp.107–141.

Beck M., Böhme H., Dziadzka M., Kunitz U., Magnus R., and Verworner D., 1998. *Linux Kernel Internals*. Addison-Wesley, Harlow, England, 2nd edn. ISBN 0-201-33143-8.

Bernstein P.A., Hadzilacos V., and Goodman N., 1987. *Concurrency Control and Recovery in Database System*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Best S., 2000. JFS Log: How the Journaled File System Performs Logging. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, edited by USENIX, pp. 163–168. USENIX, Berkeley, CA, USA. ISBN 1-880446-17-0.

Blackwell T., Harris J., and Seltzer M., 1995. Heuristic Cleaning Algorithms in Log-Structured File Systems. In *Proceedings of the Winter 1995 USENIX Conference*, pp. 277–288. USENIX Association, New Orleans (LA). ISBN 1-880446-67-7.

Cao P., Felten E.W., Karlin A.R., and Li K., 1996. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. In *ACM Transactions on Computer Systems*, 14(4):pp. 311–343. ISSN 0734-2071.

Cao P., Felten E.W., and Li K., 1994. Application-Controlled File Caching Policies. In *Proceedings of the Usenix Summer 1994 Technical Conference*, pp. 171–182. Usenix Association, Boston, MA, USA. ISBN 1-880446-62-2.

Card R., Tsó T., and Tweedie S., 1994. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*. ISBN 90-367-0385-9.

Chandra S. and Chen P.M., 1998. How Fail-stop are Faulty Programs? In *Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pp. 240–249. IEEE, Munich, Germany.

Chang F. and Gibson G.A., 1999. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 1–14. USENIX Association, New Orleans, LA. ISBN 1-880446-39-1.

Chao C., English R.M., Jacobson D., Stepanov A.A., and Wilkes J., 1992. Mime: A High Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. Rep. HPL-CSP-92-9 rev 1, Hewlett-Packard Company, Palo Alto, CA.

Chen P.M., Lee E.K., Gibson G.A., Katz R.H., and Patterson D.A., 1994. RAID: High-Performance, Reliable Secondary Storage. In *Computing Surveys*, 26(2):pp. 145–185.

Chen P.M., Ng W.T., Chandra S., Aycock C., Rajamani G., and Lowell D., 1996. The Rio File Cache: Surviving Operating Systems Crashes. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 74–83. ACM Press, Cambridge, Massachusetts.

Cheriton D.R. and Duda K.J., 1995. Logged Virtual Memory. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 26–38. ACM Press. ISBN 0-89791-715-4.

Choi J., Noh S.H., Min S.L., and Cho Y., 2002. Design, Implementation, and Performance Evaluation of a Detection-Based Adaptive Block Replacment Scheme. In *IEEE Transactions on Computers*, 51(7):pp. 793–800.

Chutani S., Anderson O.T., Kazar M.L., Leverett B.W., Mason W.A., and Sidebotham R.N., 1992. The Episode File System. In *USENIX Conference Proceedings*, pp. 43–60. USENIX, San Francisco, CA.

Comer D., 1979. The Ubiquitous B-tree. In *Computing Surveys*, 11(2):pp. 121–137.

Engler D.R., Kaashoek M.F., and O'Toole Jr. J., 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pp. 251–266. Copper Mountain Resort, Colorado.

English R.M. and Stepanov A.A., 1992. Loge: A Self-Organizing Disk Controller. In *USENIX Conference Proceedings*, pp. 237–252. USENIX, San Francisco, CA.

Ext2FS. The Ext2FS Home Page.
URL `http://e2fsprogs.sourceforge.net/ext2.html`

FreeBSD. The FreeBSD Project.
URL `http://www.freebsd.org`

Ganger G.R. and Kaashoek M.F., 1997. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pp. 1–17. USENIX Association, Anaheim (CA).

Ganger G.R., McKusick M.K., Soules C.A.N., and Patt Y.N., 2000. Soft Updates: A Solution to the Metadata Update Problem in File Systems. In *ACM Transactions on Computer Systems*, 18(2):pp. 127–153. ISSN 0734-2071.

Ganger G.R. and Patt Y.N., 1994. Metadata Update Performance in File Systems. In *Proc. First Symposium on Operating Systems Design and Implementation*, pp. 49–60.

Geist R. and Daniel S., 1987. A Continuum of Disk Scheduling Algorithms. In *ACM Transactions on Computer Systems*, 5(1):pp. 77–92.

Gifford D.K., Needham R.M., and Schroeder M.D., 1988. The Cedar File System. In *Communications of the ACM*, 31(3):pp. 288–298. ISSN 0001-0782.

Gray J., 1981. The Transaction Concept: Virtues and Limitations. In *Very large data bases: proceedings: seventh International Conference on Very Large Data Bases, Cannes, France, September 9–11, 1981*, edited by C. Zaniolo and C. Delobel, pp. 144–154. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA.

Gray J. and Reuter A., 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California. ISBN 1-55860-190-2.

Griffioen J. and Appleton R., 1994. Reducing File System Latency using a Predictive Approach. In *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, edited by USENIX Association, pp. 197–207. USENIX, Berkeley, CA, USA. ISBN 1-880446-62-6.

Grimm R., Hsieh W.C., De Jonge W., and Kaashoek M.F., 1996. Atomic Recovery Units: Failure Atomicity for Logical Disks. In *16th International Conference on Distributed Computing Systems (16th IDCS'96)*, pp. 26–35? IEEE, Hong Kong. MIT Laboratory for Computer Science, USA.

Hagmann R., 1987. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, vol. 21, pp. 155–162.

Haskin R., Malachi Y., Sawdon W., and Chan G., 1988. Recovery Management in QuickSilver. In *ACM Transactions on Computer Systems*, 6(1):pp. 82–108. ISSN 0734-2071.

Heidemann J. and Popek G., 1995. Performance of Cache Coherence in Stackable Filing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 127–142.

Heidemann J.S. and Popek G.J., 1994. File-system Development with Stackable Layers. In *ACM Transactions on Computer Systems (TOCS)*, 12(1):pp. 58–89. ISSN 0734-2071.

Hu Y., Nightingale T., and Yang Q., 2002. RAPID-Cache — A Reliable and Inexpensive Write Cache for High Performance Storage Systems. In *IEEE Transactions on Parallel and Distributed Systems*, 13(3):pp. 290–307. ISSN 1045-9219.

Hu Y. and Yang Q., 1996. DCD-Disk Caching Disk : A New Approach for Boosting I/O Performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pp. 169–178. ACM Press, New York. ISBN 0-89791-786-3.

Irlam G., 1993. UNIX File Size Survey (UFS93).
URL `http://www.base.com/gordoni/ufs93.html`

Iyer S. and Druschel P., 2001. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pp. 117–130. ACM Press. ISBN 1-58113-389-8.

JFS, 2002. The JFS Project Web Site.
URL `http://www-124.ibm.com/developerworks/oss/jfs`

Johnson T. and Shasha D., 1989. Utilization of B-trees with Inserts, Deletes and Modifies. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 235–246. ACM Press, Philadelphia (PA).

de Jonge W., Kaashoek M., and Hsieh W., 1993. The Logical Disk: A New Approach to Improving File Systems. In Proceedings SOSP 1993, pp. 15–28.

de Jonge W. and Schijf A., 1990. B-trees Reconsidered - A Comparison With W-trees. In *Proceedings of thirteenth International Seminar on Data Base Management Systems*, pp. 13–24. Mamaia, Romania.

Kaashoek M.F., Engler D.R., Ganger G.R., Briceño H.M., Hunt R., Mazières D., Pinckney T., Grimm R., Jannotti J., and Mackenzie K., 1997. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pp. 52–65. Saint-Malô, France.

Khalidi Y.A. and Nelson M.N., 1993. Extensible File Systems in Spring. In Proceedings SOSP 1993, pp. 1–14.

Kleiman S., 1986. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Conference Proceedings*, pp. 238–247. USENIX, Atlanta, GA.

Kowalski T., 1978. *FSCK — The UNIX System Check Program*. Bell Laboratory, Marray Hill, N.J. 07974.

Kroeger T.M. and Long D.D.E., 2001. Design and Implementation of a Predictive File Prefetching Algorithm. In *Proceedings of the 2001 USENIX Annual Technical Conference: June 25–30, 2001, Marriott Copley Place Hotel, Boston, Massachusetts, USA*, edited by USENIX, pp. 105 – 118. USENIX, Berkeley, CA, USA. ISBN 1-880446-09-X.

Lehman P.L. and Yao S.B., 1981. Efficient Locking for Concurrent Operations on B-trees. In *ACM Transactions on Database Systems*, 6(4):pp. 650–670. ISSN 0362-5915.

Linux Trace. Linux Disk Trace Buffer. Trace Distribution Center, Brigham Young University.
URL `http://traces.byu.edu/new/Tools/`

Linux XFS, 1999. Linux XFS.
URL `http://oss.sgi.com/projects/xfs`

Lorie R.A., 1977. Physical Integrity in a Large Segmented Database. In *ACM Transactions on Database Systems (TODS)*, 2(1):pp. 91–104. ISSN 0362-5915.

Lumb C., Schindler J., Ganger G.R., Riedel E., and Nagle D.F., 2000. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pp. 87–102. The USENIX Association, Berkeley, CA.

Lumb C.R., Schindler J., and Ganger G.R., 2002. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pp. 275–288. USENIX Association, Berkeley, CA.

Matthews J.N., Roselli D., Costello A.M., Wang R.Y., and Anderson T.E., 1997. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, SIGOPS, Saint Malo, France. ISBN 0-89791-916-5.

McKusick M.K. and Ganger G.R., 1999. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *FREENIX Track, USENIX Annual Technical Conference*, pp. 1–17.

McKusick M.K., Joy W.N., Leffler S.J., and Fabry R.S., 1984. A Fast File System for UNIX. In *ACM Transactions on Computer Systems*, 2(3):pp. 181–197.

McVoy L.W. and Kleiman S.R., 1991. Extent-like Performance from a UNIX File System. In *Proceedings of the Winter 1991 USENIX Conference*, pp. 33–43. USENIX Association, Dallas (TX).

Mullender S.J. and Tanenbaum A.S., 1984. Immediate Files. In *Software & Practice and Experience*, 14(4):pp. 365–368. ISSN 0038-0644.

Ng W.T. and Chen P.M., 2001. The Design and Verification of the Rio File Cache. In *IEEE Transactions on Computers*, 50(4):pp. 322–337.

Nightingale T., Hu Y., and Yang Q., 1999. The Design and Implementation of a DCD Device Driver for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, pp. 295–308. USENIX Association, Berkeley, CA.

Nugent J., Arpaci-Dusseau A.C., and Arpaci-Dusseau R.H., 2003. Controlling your PLACE in the File System with Gray-box Techniques. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pp. 311–324.

Ousterhout J. and Douglis F., 1989. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. In *opsysrev*, 23(1):pp. 11–28.

Ousterhout J.K., 1990. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In Proceedings USENIX 1990.

Ousterhout J.K., Cherenson A.R., Douglis F., Nelson M.N., and Welch B.B., 1988. The Sprite Network Operating System. In *IEEE Computer*, 21(2):pp. 23–36.

Ousterhout J.K., Costa H.D., Harrison D., Kunze J.A., Kupfer M., and Thompson J.G., 1985. A Trace-Driven Analysis of the Unix 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 15–24. ACM, Orcas Island WA (USA).

Patterson R.H., Gibson G.A., Ginting E., Stodolsky D., and Zelenka J., 1995. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 79 – 95. ISBN 0-89791-715-4.

Proceedings SOSP 1993, 1993. *Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*. ACM SIGOPS, ACM Press, Asheville (NC). ISBN 0-89791-632-8.

Proceedings USENIX 1990, 1990. *Proceedings of the Summer 1990 USENIX Technical Conference*. USENIX Association, Anaheim (CA).

Reiser H. ReiserFS.
URL `http://www.namesys.com`

Rosenblum M., 1992. *The Design and Implementation of a Log-structured File System*. Ph.D. thesis, U.C. Berkeley. Report UCB/CSD 92/696.

Rosenblum M. and Ousterhout J.K., 1990. The LFS Storage Manager. In Proceedings USENIX 1990, pp. 315–324.

Rosenblum M. and Ousterhout J.K., 1991. The Design and Implementation of a Log-structured File System. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pp. 1–15. ACM Press. ISBN 0-89791-447-3.

Rosenblum M. and Ousterhout J.K., 1992. The Design and Implementation of a Log-Structured File System. In *ACM Transactions on Computer Systems*, 10(1):pp. 26–52.

Rosenkrantz D.J., Stearns R.E., and Philip M. Lewis I., 1978. System Level Concurrency Control for Distributed Database Systems. In *ACM Transactions on Database Systems (TODS)*, 3(2):pp. 178–198. ISSN 0362-5915.

Santry D.S., Feeley M.J., Hutchinson N.C., Veitch A.C., Carton R.W., and Ofir J., 1999. Deciding When to Forget in the Elephant File System. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pp. 110–123. ACM Press. ISBN 1-58113-140-2.

Satyanarayanan M., Mashburn H.H., Kumar P., Steere D., and Kistler J.J., 1994. Lightweight Recoverable Virtual Memory. In *ACM Transactions on Computer Systems*, 12(1):pp. 33–57.

Schindler J., Griffin J.L., Lumb C.R., and Ganger G.R., 2002. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pp. 259–274. USENIX Association, Berkeley, CA.

Schlichting R.D. and Schneider F.B., 1983. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. In *ACM Transactions on Computer Systems (TOCS)*, 1(3):pp. 222–238. ISSN 0734-2071.

Schmidt F., 1995. *The SCSI Bus and IDE Interface: Protocols, Applications and Programming*. Addison Wesley. ISBN 0-201-42284-0.

Schmuck F. and Wyllie J., 1991. Experience with Transactions in QuickSilver. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 239–53. Association for Computing Machinery SIGOPS.

Schneider F.B., 1984. Byzantine Generals in Action: Implementing Fail-Stop Processors. In *ACM Transactions on Computer Systems*, 2(2):pp. 145–154.

Seltzer M., Bostic K., McKusick M.K., and Staelin C., 1993. An Implementation of a Log-Structured File System for UNIX. In *USENIX Technical Conference Proceedings*, pp. 307–326. USENIX, San Diego, CA. ISBN 1-880446-48-0.

Seltzer M., Chen P., and Ousterhout J., 1990. Disk Scheduling Revisited. In *USENIX Conference Proceedings*, pp. 313–324. USENIX, Washington, D.C.

Seltzer M., Smith K.A., Balakrishnan H., Chang J., McMains S., and Padmanabhan V., 1995. File System Logging versus Clustering: A Performance Comparison. In *USENIX Conference Proceedings*, pp. 249–264. USENIX, New Orleans, LA.

Seltzer M.I., 1993. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pp. 503+. Vienna, Austria.

Seltzer M.I., Ganger G.R., McKusick M.K., Smith K.A., Soules C.A.N., and Stein C.A., 2000. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pp. 71–84. USENIX Ass., Berkeley, CA.

Shriver E., Small C., and Smith K.A., 1999. Why Does File System Prefetching Work? In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, pp. 71–84. USENIX Association, Berkeley, CA.

Smith K. and Seltzer M., 1997. File System Aging – Increasing the Relevance of File System Benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurment and Modeling of Computer Systems*.

Solomon D.A. and Russinovich M.E., 2000. *Inside Microsoft®Windows®2000*. Microsoft Programming Series. Microsoft Press. ISBN 0-7356-1021-5.

Soules C.A.N., Goodson G.R., Strunk J.D., and Ganger G.R., 2003. Metadata Efficiency in Versioning File Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies FAST '03*.

Stonebraker M., 1981. Operating System Support for Database Management. In *Communications of the ACM*, 24(7):pp. 412–418. ISSN 0001-0782. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

Sweeney A., Doucette D., Hu W., Anderson C., Nishimoto M., and Peck G., 1996. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pp. 1–14. USENIX Association, San Diego (CA).

Teorey, J. T., Pinkerton, and B. T., 1972. A Comparative Analysis of Disk Scheduling Policies. In *Communications of the ACM*, 15(3):pp. 177–184. ISSN 0001-0782.

Teorey T.J., 1972. Properties of Disk Scheduling Policies. In *Proc. FJCC, AFIPS*, 41.

Tweedie S., 2000. Ext3, Journaling File System.
URL `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`

Wang J. and Hu Y., 2002. WOLF — A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File System. In *Proceedings of the FAST '02 Conference on File and Storage Technologies*, edited by USENIX, pp. 47–60. USENIX, Berkeley, CA, USA. ISBN 1-880446-03-0.

Wieringa R. and de Jonge W., 1995. Object Identifiers, Keys, and Surrogates: Object Identifiers Revisited. In *Theory and Practice of Object Sytems*, 1(2):pp. 101–114.

Worthington B.L., 1995. *Aggressive Centralized and Distributed Scheduling of Disk Requests*. Ph.D. thesis, University of Michigan.

Worthington B.L., Ganger G.R., and Patt Y.N., 1994. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 241–251. ACM Press, New York, NY, USA. ISBN 0-89791-659-X.

Yao A.C.C., 1978. On Random 2-3 Trees. In *Acta Informatica*, 9(2):pp. 159–170. ISSN 0001-5903 (print), 1432-0525 (electronic).

Index

Page numbers written in boldface indicate the page where the corresponding entry is defined or explained. An entry can occur both as a main entry (i.e., listed by itself) and as a subentry (i.e., listed under another main entry). In such a case, the subentry lists only the boldface page number. The remaining occurrences are listed under the main entry.